FACULTY OF ENGINEERING
AND ARCHITECTURE

Adaptive Semantic Stream Reasoning for Healthcare

Mathias De Brouwer

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Computer Science Engineering

Supervisors
Prof. Filip De Turck, PhD - Prof. Femke Ongenae, PhD
Department of Information Technology
Faculty of Engineering and Architecture, Ghent University

August 2023

GHENT
UNIVERSITY

ISBN 978-94-6355-741-2
NUR 984
Wettelijk depot: D/2023/10.500/73

Members of the Examination Board

Chair

Prof. Patrick De Baets, PhD, Ghent University

Other members entitled to vote

Prof. Jean-Paul Calbimonte Pérez, PhD, Haute Ecole Spécialisée de Suisse occidentale, Switzerland

Femke De Backere, PhD, Ghent University

Prof. Dieter De Witte, PhD, Ghent University

Prof. Anastasia Dimou, PhD, KU Leuven

Supervisors

Prof. Filip De Turck, PhD, Ghent University

Prof. Femke Ongenae, PhD, Ghent University

Dankwoord

"The future's in the hands of you and me, so let's all get together, we can all be free, spread love and understanding, positivity"

– Ed Sheeran, What Do I Know? (2017), from the ÷ (Divide) album

Om dit dankwoord te openen, had ik eigenlijk geen andere keuze dan dit te doen met een songquote van Ed Sheeran, met voorsprong mijn favoriete artiest op aarde. We schrijven 2017, het jaar waarin het album ÷ op de wereld werd losgelaten, en ook het jaar waarin ik enkele maanden later in augustus met de start van mijn doctoraat een nieuw hoofdstuk van mijn leven indook. Ik had toen nog geen flauw idee dat ik de naam van dit album, ÷ of dus 'divide', zou gebruiken voor één van de belangrijkste componenten in mijn doctoraatsonderzoek, en dat deze naam daardoor maar liefst exact 1351 keer in dit boek zou voorkomen. Maar hier zijn we dan, augustus 2023, een goeie zes jaar later, en zo geschiedde.

De quote hierboven komt uit een lied op het ÷ album waarvan de boodschap meerdere lagen heeft, maar tegelijkertijd heel simpel is: aan de basis van het leven staan waarden als liefde, vriendschap, begrip voor elkaar, en positiviteit. Het klinkt simpel en evident, maar wordt soms sneller over het hoofd gezien dan je zou willen. Desondanks is het echter een krachtige visie, die me mee heeft geholpen om dit doctoraat tot een goed einde te brengen. Onvermijdelijk was het een hectische rit om tot op dit punt te geraken, waar het geregeld wel eens even wat minder ging op vlak van studies of doctoraat. Het besef dat er dan altijd belangrijkere dingen zijn om op terug te vallen, met name familie en vrienden, hielp enorm om dit te relativeren en nadien weer met hernieuwde moed verder door te zetten. Het heeft me andermaal geleerd wat het belang is van deze waarden steeds in je achterhoofd te houden.

Alles tezamen vormt dit boek het orgelpunt van een periode van zes jaar als onderzoeker aan de IDLab onderzoeksgroep, of bij uitbreiding van de elf jaar waarin ik reeds actief ben aan de UGent. Dat klinkt als een eeuwigheid, en dat is het eigenlijk ook. Ik zie mezelf nog steeds toekomen in Gent op de Campus Sterre op de eerste dag van mijn allereerste academiejaar in 2012, volledig overweldigd door het universitaire reilen en zeilen. Het was al verre van een evidentie om de juiste studierichting te kiezen toen, laat staan dat ik dus op dat moment nog maar het geringste idee had dat ik ooit een doctoraat in de computerwetenschappen zou indienen. Zoals

met alles in het leven is het een aaneenschakeling van bepaalde keuzes geweest die me hiertoe hebben gebracht, een gevolg van het welgekende butterfly effect op een mensenleven. Had ik na één week op de Sterre niet beslist dat wiskunde precies toch niet de ideale opleiding was voor mij en informatica misschien beter geschikt was ... had ik de wiskundekriebel toch opnieuw gevolgd die kwam opborrelen toen de master wiskundige informatica zich aanbood op het kruispunt na de bacheloropleiding informatica ... had ik destijds die interessante masterproef binnen de gezondheidszorg niet gekozen met Filip en Femke als promotoren ... dan stond ik allicht vandaag niet in deze positie. Maar hier zijn we nu toch uiteindelijk beland, en ik ben blij met de gemaakte keuzes die daarvoor hebben gezorgd. En ik zou liegen als ik zou zeggen dat ik niet toch ook een beetje trots ben op dit orgelpunt.

Tijdens de voorbij zes jaar ben ik met een grote hoeveelheid mensen in contact gekomen, binnen en buiten de virtuele muren van de academische wereld. Ik wil van dit dankwoord dan ook gebruik maken om velen onder hen te bedanken. Indien je dit dankwoord echter leest en je jezelf nergens expliciet of impliciet vermeld ziet, ondanks dat je dit toch ergens verwachtte, wil ik me alvast excuseren. Ik heb hieronder een welgemeende poging gedaan om zo inclusief mogelijk te zijn, maar het is bijna onmogelijk om geen mensen te vergeten. Daarom, in dat geval, ook bedankt aan jou.

Om te beginnen wil ik graag diegenen bedanken die ervoor gezorgd hebben dat ik dit doctoraat heb kunnen en mogen doen, met name mijn promotoren Filip De Turck en Femke Ongenae. Filip, bedankt om in mij te geloven en om mij de kans aan te bieden om aan dit doctoraat te starten tijdens het uitvoeren van mijn masterpoef in mijn laatste masterjaar. Ik herinner me nog levendig hoe je mailtje daarover toekwam in de kerstperiode van 2016 en hoe ik meteen enthousiast van mijn kamer naar beneden liep om mijn ouders in te lichten. Bedankt voor je begeleiding doorheen de voorbije jaren, voor de relevante adviezen en feedback op mijn papers en ander werk doorheen de jaren, en voor de fijne samenwerking in het algemeen. Femke, bedankt voor alles. Bedankt voor de eindeloos vele (en soms eindeloos lange) follow-up meetings, die geregeld flexibel rond andere meetings ingepland werden en soms in meerdere delen moesten worden opgesplitst. Ik ben altijd onder de indruk geweest van hoe snel jij telkens mee bent met het werk waar ik actief mee bezig ben, wetende dat je dit voor ettelijke doctoraatsstudenten doet naast ook al je ander werk. Bedankt voor al je tips, feedback, ideeën, en je bereidheid om eender wanneer op al mijn vragen te antwoorden, klein of groot. Bedankt daarnaast om tussen al het harde werken door ook steeds jezelf te zijn, waarbij er altijd ruimte was voor een (stevige) lach of babbel. Ik denk oprecht niet dat iemand zich een betere begeleider/copromotor kan wensen voor hun doctoraat, dus een gigantische dankjewel daarvoor.

Ook wil ik graag mijn juryleden, Patrick De Baets, Dieter De Witte, Femke De Backere, Anastasia Dimou en Jean-Paul Calbimonte, bedanken om de tijd te nemen om mijn langer dan gemiddeld boek te lezen en evalueren, voor de constructieve feedback en suggesties, en voor de interessante discussies tijdens de interne verdediging.

Naast mijn promotoren en juryleden wil ik ook graag de IDLab onderzoeksgroep bedanken om het uitvoeren van mijn onderzoek en voltooien van mijn doctoraat te faciliteren. Een specifieke dankjewel daarvoor aan Piet Demeester. Ook bedankt aan alle mensen binnen IDLab die ervoor zorgen dat alles op administratief, logistiek en technisch gebied op wieltjes loopt. Bedankt dus aan Martine Buysse, Davinia Stevens, Karen Van Landeghem, Bernadette Becue, Vicent Borja-Torres, Brecht Vermeulen, Joeri Casteels, Sai Roberts, en de anderen. Bedankt ook aan Sabrina en collega's om ervoor te zorgen dat we steeds in een propere omgeving mochten komen werken.

Doorheen de voorbije zes jaar ben ik betrokken geweest bij verschillende onderzoeksprojecten, waarbij ik het genoegen heb gehad om met vele industriepartners uit verschillende bedrijven te mogen samenwerken. Daarom wil ik graag volgende mensen bedanken voor de fijne en vruchtbare samenwerking: Pieter Crombez, Wim Dereuddre en Piet Verheye van Televic Healthcare, Koen Casier en Jan Van Ooteghem van Amaron, Joeri Ruyssinck en Joachim van der Herten van ML2Grow, Julie Wyffels en Naomi Verbeke van Z-Plus, Luk Overmeire van VRT, Erwin Cornelis en Ramses Zeulevoet van Rombit, Ward Vande Capelle van Energy Lab, en Nathalie De Mey en Thomas Godon van het vroegere Videohouse.

Vervolgens zou ik graag iedereen willen bedanken die het mogelijk heeft gemaakt dat ik zes academiejaren lang het immens boeiende jaar- en projectvak 'Design Project' heb mogen begeleiden. Onderwijs heeft me altijd al sterk geboeid, en ik herinner me nog goed hoe ik op het allereerste gesprek met Femke en Filip over een mogelijk doctoraat meteen aangaf dat de optie om betrokken te zijn bij universitair onderwijs voor mij een belangrijk element was in de keuze voor het doctoraat. Jullie gaven dan ook meteen aan dat dit mogelijk was, en voor ik het wist was de trein vertrokken, waarvoor dank. Met momenten vroeg de begeleiding van dit vak veel van mijn tijd, kostbare tijd die gedeeld moest worden met mijn ander, onderzoeksgerelateerde werk. Filip en Femke Ongenae, bedankt om desondanks toch te faciliteren om, in de mate van het mogelijke, me toe te laten om hierin mijn ei kwijt te kunnen. Deze afwisseling gaf me steeds de nodige energie om me volop op mijn onderzoek en doctoraat te blijven focussen. Naast Filip wens ik ook graag de andere fijne (ex-)collega's te bedanken waarmee ik bij de begeleiding van dit vak heb mogen samenwerken en van wie ik veel heb geleerd: Dirk Stroobandt, Frank Gielen, Jelle Nelis, Anna Lin, Laurens Martin, Sam Van Damme, Stéphanie Carlier, Joris Heyse, en tot slot Femke De Backere. Joris, Stéphanie, en Femke De Backere: het was fijn om naast onze bureau ook de begeleiding van dit vak met jullie te delen. Dit zorgde voor een extra gemeenschappelijk element wat geregeld leidde tot spontane discussies op bureau over het vak. Algemeen gezien zorgde de aangename sfeer onder de volledige teaching staff ervoor dat ik steeds met plezier dit vak heb begeleid. Bedankt dus ook daarvoor. Femke De Backere, een bijzondere vermelding voor jou is zeker op zijn plaats. De energie en tijd die jij in dit vak steekt, is van een ongeziene hoogte, en bewonder ik enorm. Bedankt om me mee op sleeptouw te nemen in de eerste jaren voor de begeleiding van het vak, bedankt voor de energie die je ook afstraalt op de rest van de teaching staff, en bedankt voor het vertrouwen dat je in de assistenten stelt om het vak mee te mogen dragen.

Vervolgens wil ik ook een welgemeende dankjewel betuigen aan de fijne collega's waarmee ik doorheen de jaren bureau 200.009 in iGent gedeeld heb. Sommigen onder hen kwamen reeds aan bod, maar nog niet iedereen, dus vandaar som ik ze graag nog even op. Dus, bedankt aan Femke Ongenae, Femke De Backere, Bram Steenwinckel, Joris Heyse, Stéphanie Carlier, Pieter Bonte, Stijn Verstichel, Gilles Vandewiele, Philip Leroux, Kyana Bosschaerts, Michael Weyns, en Ziye Fang. Samen zorgden we steeds voor een gezellige werksfeer. Ik denk met de glimlach terug aan de vele dagen, vooral dan pre-corona in de eerste jaren van mijn doctoraat, waarin het vaak bakje vol was op bureau, en waarin er altijd wel iets te beleven viel. Ook nu kom ik uiteraard nog steeds graag naar bureau, en is het altijd een fijne bedoening. Jullie zorgden en zorgen er allemaal samen voor dat het steeds fijn is om een gezonde balans te houden tussen thuiswerk en werken op bureau. Bedankt daarvoor. Bij uitbreiding ook bedankt aan de andere leden van het knowledge management and reasoning team.

Graag wil ik ook nog enkele collega's van onze bureau in iGent iets concreter bedanken. Stijn Verstichel, bedankt om me in 2017 op sleeptouw te nemen op mijn allereerste conferentie in Wenen. Alles voelde nieuw en onwennig, maar jouw verhalen, inzichten en begeleiding zorgden ervoor dat ik met een open blik leerde kijken naar de academische wereld en de wondere wereld der conferenties. Pieter Bonte, bedankt om me vooral in de eerste jaren van mijn doctoraat bij te staan en op weg te helpen met je spot-on technische kennis en inzicht in ons onderzoeksdomein. Bram Steenwinckel, bedankt voor alle hulp en steun bij het PROTEGO project. Ik herinner me nog levendig hoe we in en rond het HomeLab van de ene verbazing in de andere zijn blijven vallen. Telkens wanneer we dachten dat we het laatste nu wel gezien hadden, volgde er toch nog een nieuwe overtreffende trap. Het feit dat we steeds konden terugvallen op elkaar om te debuggen, bespreken, en onze ontreddering delen, geldt desondanks toch als een fijne herinnering.

Daarnaast mag ik me sinds ruim een jaar ook lid noemen van het PreDiCT onderzoeksteam. Sofie Van Hoecke, heel erg bedankt om hiervoor het initiatief te nemen en voor de energie die je in dit team steekt. Ik ben niet de grootste fan van verandering van routine en vond het dan ook wat gek om plots een tweede bureau te hebben in de AA toren, maar al snel voelde ik me daar helemaal op mijn gemak. Ik ben blij om deel van het team uit te maken en om via deze weg heel wat nieuwe toffe collega's erbij te hebben. Bedankt dus ook aan hen allemaal. Een specifieke dankjewel ook aan de PreDiCT-collega's van de PROTEGO en mBrain projecten, waarmee ik bijzonder intensief heb samengewerkt bij het ontwikkelen (en eindeloos debuggen) van verscheidene proof-of-concepts, demo's en apps, zoals Marija Stojchevska, Pieter Moens, Jonas Van Der Donckt, Stef Pletinck, Nathan Vandemoortele, Bram Steenwinckel, en anderen. Ondanks de issues die onvermijdelijk wel eens opdoken in dergelijke projecten, was het toch steeds fijn om met jullie samen te werken en wanneer nodig naar de optimale oplossing te zoeken. Specifiek sprekend over het mBrain project wil ik ook graag Nicolas Vandenbussche en Koen Paemeleire van het UZ Gent bedanken voor de fijne en uitermate boeiende samenwerking gedurende de voorbije jaren.

Zoals vermoedelijk al duidelijk is na de voorbije paragrafen, heb ik doorheen de jaren het genoegen gehad om te mogen samenwerken met heel wat fijne onderzoekers en onderzoeksmedewerkers. Een ruim aantal onder hen kwam dus reeds aan bod in dit dankwoord, maar ik wens ook zeker graag alle anderen te bedanken die ik nog niet eerder expliciet vermeld heb. Vandaar, ook aan Tom Windels, Stephanie Chen, Jeroen Van Der Donckt, Emile Deman, Vic Degraeve, Dörthe Arndt, Maria Torres Vega, Jerico Moeyersons, Matthias Strobbe, Sai Roberts, Christof Mahieu, Myriam Sillevis Smitt, Wannes Kerckhove, Jasper Vaneessen, Thomas Dupont, Bruno Volckaert, Julián Andrés Rojas Meléndez, Ruben Taelman, Ruben Verborgh, Miel Vander Sande, Joachim Van Herwegen, Pieter Colpaert, Ben De Meester, Pieter Heyvaert, Brecht Van de Vyvere, Sven Lieber, Glenn Daneels, Esteban Municio, Bart Braem, Jeroen Famaey, Steven Latré, Davy Preuveneers, Majid Makki, An Jacobs, Tom Seymoens, Annelies Goris, Wouter Durnez, en de anderen die ik hier nu ongetwijfeld vergeten ben: bedankt voor de fijne samenwerking. Bedankt ook aan velen onder hen en heel wat eerder vermelde collega's voor het aangename gezelschap op verschillende conferenties doorheen de voorbije jaren, van Lyon naar Portorož over Wenen tot zelfs in Auckland in Nieuw-Zeeland.

Na de bedanking van dit ruime arsenaal aan collega's en mensen waarmee ik op professioneel gebied heb mogen samenwerken, is het tijd om mij te richten op mijn leven buiten de muren van de universiteit. Ook hier wil ik heel wat mensen bedanken.

Eerst, vooraleer ik verderga: dankjewel aan Rik voor de mooie jaren die we samen vanaf de start van mijn doctoraat beleefden. Ook een bijzondere dankjewel aan Hans, Anja, Annita, Owen, Kokie en Louisken.

Over dan naar de mensen zonder wie dit doctoraat ook verre van mogelijk zou geweest zijn: mijn familie en vrienden. Het is altijd gevaarlijk en oneerlijk om aan een lijstje hiervan te beginnen, aangezien dit er onvermijdelijk toe leidt dat je mensen vergeet of onvoldoende belicht. Anderzijds ga ik er ook ergens vanuit dat zij die tot deze categorie behoren, dit ongetwijfeld zelf wel beseffen, dus vandaar, in het algemeen aan jullie allemaal: een dikke merci! Desalniettemin kan ik niet anders dan een poging wagen om bepaalde mensen en groepen toch ook specifiek te bedanken. Hier gaan we.

Laura, bedankt om de vriendin te zijn die je bent. Je bent de verpersoonlijking van wat een beste vriendin hoort te zijn. Onze vriendschap is op zijn zachtst gezegd uniek en is door de jaren heen onbreekbaar gebleken. Nathalie, bedankt voor de mooie vriendschap, ettelijke Foubert-avonden en fijne babbels, en ook ellenlange berichten tussendoor waarmee je gemakkelijk een halfuur zoet was om ze te beantwoorden. Laurens Van Hoye, naast een fijne (ex-)collega kan ik je vooral een goeie vriend noemen. Iemand waarmee ik vanaf die eerste dag van het academiejaar in 2012 zo ongeveer exact hetzelfde traject heb gedeeld, en dat schept een band. Bedankt onder andere voor de oneindig vele gezellige treinritten en lange video calls, reeds van in het prille begin van onze studies tot nu. Jens, Tim en Nick, ook bedankt voor de fijne verderzetting van onze goeie vriendschap na onze studies, inclusief steeds gezellige bijeenkomsten om

de zoveel tijd met obligatoir natje en droogje, en de citytripjes. Here's to many more! Boys oorspronkelijk van 't plein in Sint-Niklaas al die jaren (namen noemen hier is per definitie mensen vergeten, maar wie tot de bende behoort herkent zichzelf uiteraard), bijna exact hetzelfde kan eigenlijk zo ongeveer gezegd worden over jullie. Thanks daarvoor! Jonathan, bedankt voor het hernieuwen van onze vriendschap en het samenbrengen van fijne mensen, en een gigantische dankjewel voor al die eindeloze, goeie gesprekken. Ook bedankt aan alle andere goeie vrienden, waaronder Steven & Anske voor de geslaagde samenkomsten op allerlei locaties waarin we – naast andere zaken, en vaak vergezeld van een goed glas – onze passie en verhalen over reizen uitwisselden, Xavier voor de jarenlange vriendschap, Kirsten voor de altijd gezellige afspraakjes, en (zoals al zo vaak gezegd) de anderen die ik hier nu allicht vergeet.

Voor ik overga naar mijn familie, las ik ook met plezier een intermezzo in. Hierin wens ik graag nog een opsomming te geven van andere zaken en personen die ik dankbaar ben, en zonder wie of wat het voltooien van mijn doctoraat ook een pak minder aangenaam zou zijn geweest. Voor vele lezers kan dit dan misschien ook als een zeer willekeurige lijst overkomen, en dat is het zeker ook, maar alle elementen hebben toch een kleine tot soms zeer grote rol gespeeld. Vandaar, bedankt aan Ed Sheeran, Arsenal Football Club (whatever the weather), Lotus Bakeries (en de uitvinder van de Zebra!), Rafa Nadal, onze Lonely Planet (of Pachamama) en zijn prachtige locaties, De Mol, de wondere wereld der patisserie, Big Brother (jawel), FPL, F.C. De Kampioenen (opnieuw jawel!), en eigenlijk nog zoveel meer.

Zoals aangekondigd richt de laatste halte van dit dankwoord zich tot mijn familie. Uiteraard wil ik iedereen daarvan bedanken voor de warme omgeving waarin ik ben opgegroeid, inclusief alle tantes en nonkels, neven en nichten. Maar de bijzonderste dankjewel gaat uiteraard naar mijn dichtste familie.

Zoals ik al eerder zei, voelt dit doctoraat aan als een orgelpunt. Maar niet enkel van de laatste zes jaar. Oma, Peter, Meme, Pepe, ik denk nog vaak aan jullie en de wijze lessen die ik doorheen de jaren van jullie meekreeg. Ik zou niet staan waar ik vandaag sta zonder de warme thuis en geborgen jeugd die ik gekend heb. Daar hebben jullie een gigantisch grote rol in gespeeld. Ik koester de mooie herinneringen aan de ontelbare bezoekjes en familiebijeenkomsten. Ik hoop dat jullie trots zijn.

Bieke en Tom, als kleine jongere broer zag ik jullie reeds het studentenleven van Gent intrekken terwijl ik zelf nog op de lagere en middelbare schoolbanken zat. Ik zag hoe jullie elk succesvol jullie weg vonden in het hoger onderwijs, en met succes een universitaire richting afrondden. Dit heeft mij geïnspireerd om als kleine jongen dit ook te willen bereiken, en te geloven dat ik dat ook zou kunnen. Daarnaast wil ik jullie bedanken voor de toffe jeugd die we samen beleefden. Ontelbaar vele mooie reizen maakten we samen, met als hoogtepunt voor het ganse gezin denk ik toch wel de reis naar de westkust van de USA. Prachtige herinneringen zijn dat. Bieke, doorheen de jaren denk ik dat onze band alleen maar sterker is geworden. Bedankt

onder andere om van kinds af aan altijd al als grote zus over mij te waken, en bedankt voor je luisterend oor en goeie raad. Bedankt ook voor je enthousiasme en om mij steeds op sleeptouw te nemen om nieuwe, leuke dingen te gaan doen. Stefanie, wat een zalige schoonzus ben jij. Bedankt om steeds je (h)eerlijke zelf te zijn, en om onze familiebijeenkomsten altijd net dat tikkeltje geanimeerder te maken. Het is prachtig om te zien hoe je met Bieke matcht, en wat een powerkoppel jullie zijn waar ik altijd bij terecht kan. Tom, bedankt onder andere om als grote broer ook steeds het beste met mij voor te hebben, om steeds je enthousiasme over allerlei onderwerpen met mij te delen, en om steeds klaar te staan met je uitgebreide kennis over van alles en nog wat als er een specifiek probleem is.

Bieke & Stefanie, intussen reeds meer dan drie jaar mag ik mezelf de trotse peter/nonkel noemen van jullie twee prachtige boys. Miller & Miro, jullie beseffen het allicht nog niet echt, maar jullie betekenen veel voor mij. Jullie ongeremd enthousiasme doet al mijn zorgen steeds als sneeuw voor de zon verdwijnen, en het is dan ook altijd zalig om tijd met jullie door te brengen. Ik ben oprecht gelukkig deel van jullie leven te mogen uitmaken, en ik kijk er naar uit om jullie van dichtbij te zien opgroeien de komende jaren. Weet alvast dat jullie altijd en overal op mij zullen kunnen rekenen!

Tot slot wil ik graag mijn ouders bedanken. Mama, Papa, uit de grond van mijn hart, bedankt voor alles. Bedankt voor alle kansen die jullie mij gegeven hebben. Bedankt voor jullie opvoeding waarin jullie mij de normen en waarden meegaven die belangrijk zijn in het leven. Bedankt voor de warme thuis die jullie tot op heden steeds gecreëerd hebben. Ik ben oprecht blij dat de coronaperiode achter ons ligt, en dat ik zonder schroom zo vaak bij jullie kan langskomen als ik wil. Bedankt voor de steun die ik steeds van jullie krijg op alle vlakken. Weet dat ik jullie graag zie. Ik zou nog veel meer kunnen schrijven, maar ik denk dat dat niet nodig is voor jullie om te weten hoe dankbaar ik jullie ben. Hopelijk zijn ook jullie trots.

Gent, augustus 2023
Mathias De Brouwer

Table of Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

A

AAL Ambient Assisted Living

ABox Assertional Box

ACCIO Ambient-aware provisioning of Continuous Care for Intramural Organizations

API Application Programming Interface

AR Activity Recognition

ASP Answer Set Programming

B

BAN Body Area Network

BLE Bluetooth Low Energy

BNF Backus-Naur Form

BRS Back-end Reasoning Service

BVP Blood Volume Pulse

C

C-SPARQL Continuous SPARQL

CAS Cranial Autonomic Symptoms

CEP Complex Event Processing

CH Cluster Headache

CP Continuous Processing

CPU Central Processing Unit

CSV Comma-Separated Values

D

DAHCC Data Analytics for Health and Connected Care

DDR Double Data Rate

DL Description Logics

DSMS Data Stream Management System

E

e.g. exempli gratia – for example

EDA ElectroDermal Activity

EEP Execution-Executor-Procedure

eHealth electronic Healthcare

EHR Electronic Health Record

ELE Enhanced Living Environments

F

FWO Research Foundation – Flanders (Dutch: Fonds Wetenschappelijk Onderzoek – Vlaanderen)

G

GBD Global Burden of Disease

GDPR General Data Protection Regulation

GP General Practitioner

GPS Global Positioning System

GSR Galvanic Skin Response

GUI Graphical User Interface

H

HR Heart Rate

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

I

i.e. id est – that is (in other words)

IBI Inter Beat Interval

ICHD-3 the International Classification of Headache Disorders, Third Edition

ICT Information and Communication Technology

ID IDentifier

IEEE Institute of Electrical and Electronics Engineers

IHS International Headache Society

IoT Internet of Things

IQR InterQuartile Range

IRI Internationalized Resource Identifier

IT Information Technology

J

JAR Java ARchive

JSON JavaScript Object Notation

JSON-LD JavaScript Object Notation for Linked Data

K

KB Knowledge Base

L

LARS Logic-based framework for Analytic Reasoning over Streams

LOV Linked Open Vocabularies

LoWPAN Low-power Wireless Personal Area Network

LRS Local Reasoning Service

LSD Long Slow Distance

M

MAC Media Access Control

MIDAS MIgraine Disability Assessment Test

ML Machine Learning

MSQ Migraine-Specific quality-of-life Questionnaire

N

N3 Notation3

O

OBDA Ontology-Based Data Access

OBU Observation Unit

oNCS ontology-based Nurse Call System

OWL Web Ontology Language

OWL 2 DL OWL 2 Description Logics

OWL 2 EL OWL 2 Existential quantification Logic

OWL 2 QL OWL 2 Query Language

OWL 2 RL OWL 2 Rule Language

P

PAS Personal Alarm System

PoC Proof-of-Concept

PPG PhotoPlethysmoGram

R

R2RML RDB to RDF Mapping Language

RAM Random Access Memory

RAT Reasoning About Time

RCH Research CHallenge

RCO Research COntribution

RDF Resource Description Framework

RDFS RDF Schema

REST API REpresentational State Transfer Application Programming Interface

RH Research Hypothesis

RL Rule Language

RML RDF Mapping Language

ROT Reasoning Over Time

RSEP RDF Event Processing

RSP RDF Stream Processing

RSPS RDF Stream Processing Service

RTT Round-Trip Time

Rx rate rate of Received data

S

SAREF Smart Applications REFerence

SAREF4BLDG SAREF extension for Building

SAREF4EHAW SAREF extension for the eHealth Ageing Well domain

SAREF4WEAR SAREF extension for Wearables

SBO Strategic Basic Research (Dutch: Strategisch BasisOnderzoek)

SD Standard Deviation

SF-20 MOS Short-Form General Health Survey

SNOMED CT Systematized Nomenclature of Medicine Clinical Terms

SOA Service Oriented Architecture

SOSA Sensor, Observation, Sample and Actuator

SPARQL SPARQL Protocol and RDF Query Language

SR Stream Reasoning

SSN Semantic Sensor Network

T

TBox Terminological Box

TDMA Time Division Multiple Access

TTH Tension-Type Headache

Turtle Terse RDF Triple Language

Tx rate rate of Transmitted data

U

UC Use Case

UI User Interface

URL Uniform Resource Locator

V

VKG Virtual Knowledge Graph

W

W3C World Wide Web Consortium

WHO World Health Organization

WSN Wireless Sensor Network

X

XML eXtensible Markup Language

XSD XML Schema Definition

Y

YAML YAML Ain't Markup Language

Samenvatting
– Summary in Dutch –

Het internet der dingen (ook *Internet of Things* of IoT) wordt gekenmerkt door verschillende sensoren, apparaten, actuatoren en andere 'dingen' die verbonden zijn met het internet en voortdurend gegevens genereren en verwerken in verschillende toepassingsgebieden. De gezondheidszorg is één van die toepassingsgebieden die door het IoT getransformeerd zijn. Tegen 2030 zal de wereldwijde marktomvang van het IoT in de gezondheidszorg naar verwachting meer dan verdrievoudigen ten opzichte van dit jaar tot 960 miljard dollar. Het IoT is op talrijke manieren geïntegreerd in de gezondheidszorg: patiënten worden gemonitord met wearables, terwijl hun ziekenhuiskamers of slimme huizen kunnen worden uitgerust met omgevingssensoren, lokalisatieapparaten, bewegingsdetectoren, apparaten die de toestand van elektrische toestellen controleren, en nog veel meer. Al deze apparaten genereren voortdurend stromen van realtime, ruwe gegevens. Vaak zijn er ook IoT-apparaten aanwezig die op de gedetecteerde gegevens kunnen reageren door bijvoorbeeld de automatische gordijnen te sluiten of de verwarming te regelen.

De door IoT-sensoren gegenereerde ruwe gegevens zijn betekenisloos op zichzelf. In veel toepassingsgebieden van het IoT, waaronder de gezondheidszorg, is echter veel domeinkennis en contextuele informatie beschikbaar die relevant kan zijn voor bepaalde toepassingen. Domeinkennis in de gezondheidszorg omvat medische kennis over ziekten, behandelingen en mogelijke alarmerende situaties. Contextinformatie bestaat uit het medische profiel van patiënten (zoals bijvoorbeeld het elektronisch patiëntendossier van het ziekenhuis of de huisarts), een beschrijving van de geïnstalleerde sensoren in patiëntenkamers, informatie omtrent de locaties en beschikbaarheden van zorgverleners, en nog veel meer. De beschikbaarheid van deze gegevens biedt de mogelijkheid om de stromen van sensorgegevens te integreren met de domeinkennis en contextinformatie, om daaruit realtime inzichten af te leiden over de toestand en de omgeving van de patiënt. Op deze manier kunnen contextgevoelige toepassingen ontstaan. Men kan zelfs een stap verder gaan naar bruikbare inzichten, waarbij acties worden gekoppeld aan de gegenereerde inzichten. Zo zou bijvoorbeeld de dichtstbijzijnde verpleegkundige kunnen worden gewaarschuwd om de patiënt te gaan controleren wanneer een alarmerende situatie wordt gedetecteerd.

Het integreren van IoT-gegevens met domeinkennis en contextinformatie om realtime inzichten te genereren is een uitdagende taak. Dit komt voornamelijk door twee redenen: de grote heterogeniteit van de gegevens en de hoge snelheid waarmee de gegevens worden geproduceerd. De uitdaging omtrent heterogeniteit verwijst naar

de verschillende formaten en representaties (zoals bijvoorbeeld JSON of SQL) van de gegevens in de datastromen die door IoT-sensoren en -apparaten worden gegenereerd, in de beschikbare domeinkennis en in contextinformatie. Deze heterogene aard van de gegevens bemoeilijkt de integratie ervan. De uitdaging omtrent snelheid verwijst naar de enorme snelheid van de stromen van IoT-gegevens die de verschillende sensoren en apparaten voortdurend genereren. In toepassingen in de gezondheidszorg kan deze snelheid oplopen tot meer dan 100 sensorwaarnemingen per seconde per patiënt. Bijgevolg is er nood aan efficiënte technieken om deze datastromen te verwerken.

Kennisgedreven systemen vormen een mogelijke oplossing voor de vermelde uitdagingen. In dergelijke systemen wordt de uitdaging omtrent heterogeniteit opgelost door gebruik te maken van semantiek. Semantiek laat toe om alle gegevens semantisch te verrijken en te integreren in een uniform formaat dat door computersystemen geïnterpreteerd kan worden. Technologieën van het semantische web zijn een reeks aanbevolen technologieën die deze verrijking uitvoeren met behulp van ontologieën, die alle concepten en hun relaties in een domein formeel beschrijven. Semantische redeneertechnieken kunnen dan worden gebruikt om nieuwe kennis, waaronder bijvoorbeeld bruikbare inzichten, af te leiden uit de semantische gegevens met behulp van de definities in ontologieën.

Om de uitdaging rond snelheid aan te pakken, focust een specifiek onderzoeksgebied zich op het toepassen van semantische redeneertechnieken voor datastromen, door ze op te nemen in componenten die specifiek instaan voor de verwerking van dergelijke datastromen. Dergelijke componenten evalueren voortdurend semantische query's op datastromen. Dit doen ze door windows (vensters) van een bepaalde grootte bovenop de datastromen te plaatsen. De frequentie waarmee dit gebeurt bepaalt de uitvoeringsfrequentie van de query. Bij het gebruik van semantische redeneertechnieken wordt de computationele complexiteit van het redeneren bepaald door het vereiste niveau van expressiviteit. In complexe toepassingsgebieden van het IoT, zoals de gezondheidszorg, zit de bestaande domeinkennis vervat in grote en complexe ontologieën, waardoor een hoge expressiviteit van redeneren vereist is. In het onderzoeksgebied van semantisch redeneren op datastromen bestaat er bijgevolg een trade-off omtrent performantie tussen de expressiviteit van het semantisch redeneren enerzijds en de snelheid van de data anderzijds: hoe hoger de vereiste expressiviteit is, des te lager de snelheid van de data mag zijn opdat deze op een performante manier in real time kan worden verwerkt.

Een realistische set-up van een IoT-toepassing bestaat uit een volledig netwerk van IoT-apparaten, met verschillende lokale delen in het netwerk. In een toepassing in de gezondheidszorg stemmen deze lokale delen overeen met de lokale omgevingen van de verschillende patiënten. Om de trade-off omtrent performantie bij het semantisch redeneren op datastromen aan te pakken, voeren bestaande semantische IoT-platformen de continue query's, die de datastromen met hoge snelheid moeten kunnen verwerken, voornamelijk uit op hoogwaardige servers met hoge specificaties in de centrale delen van het netwerk, of beperken zij de complexiteit van het semantisch redeneren en de query's die kunnen worden uitgevoerd. Gecentraliseerde systemen vertonen echter meerdere tekortkomingen die in strijd zijn met belangrijke vereisten in toepassingen binnen de gezondheidszorg en andere toepassingsgebieden van het IoT.

Aangezien zij alle stromen van ruwe sensordata over het netwerk naar de centrale servers sturen, bieden ze geen flexibiliteit in het beheer van de privacy van de gegevens van gebruikers. Aangezien de lokale apparaten in het netwerk niet meewerken aan de verwerking van de datastromen, hebben zij geen lokale autonomie om bepaalde bruikbare inzichten af te leiden en daarop zelf actie te ondernemen. Bovendien belasten dergelijke systemen voortdurend de netwerkcapaciteit en alle beschikbare serverbronnen voor het voortdurend doorsturen en verwerken van alle gegevens, waardoor de performantie en responsiviteit van het systeem verder worden beperkt.

Om daarom af te stappen van gecentraliseerde systemen werd het concept van trapsgewijs semantisch redeneren voorgesteld. De visie hierachter is het bouwen van een ketting van semantische redeneercomponenten voor datastromen. In het begin van de ketting worden datastromen met een hoge snelheid verwerkt door redeneercomponenten met een lage expressiviteit. De daaropvolgende redeneercomponenten kunnen de expressiviteit van het redeneren geleidelijk aan verhogen naarmate de snelheid van de data bij elke component in de ketting afneemt. Deze visie sluit perfect aan bij de visie van edge computing, waarbij heterogene IoT-apparaten met lage specificaties in de lokale randdelen van netwerken bij de gegevensverwerking worden betrokken.

De reeds bestaande visie van trapsgewijs semantisch redeneren is nog niet gerealiseerd in een generiek semantisch framework dat gedistribueerd is over een volledig netwerk, dat eenvoudig gebruikt kan worden binnen toepassingsgebieden van het IoT zoals de gezondheidszorg, en dat de vermelde tekortkomingen van gecentraliseerde systemen aanpakt. Daarom is de eerste contributie van dit proefschrift de realisatie van een dergelijk framework. Het framework heeft een generiek ontwerp dat eindgebruikers toelaat om een toepassingsspecifiek netwerk van kettingen van semantische redeneercomponenten voor datastromen in een IoT-netwerk te bouwen en configureren. Het ontwerp van het framework richt zich specifiek op twee aspecten: het verbeteren van de algemene performantie van het semantisch redeneren op datastromen voor IoT-toepassingen, en het introduceren van lokale autonomie door de lokale randapparaten van het netwerk te benutten in de ketting van redeneercomponenten. Zo kunnen bruikbare inzichten op een responsieve manier uit de datastromen worden afgeleid. Dit wordt aangetoond voor een toepassing in de gezondheidszorg waarbij patiënten in ziekenhuiskamers worden gemonitord: wanneer een alarmerende situatie zich voordoet, kan de meest geschikte verpleegkundige om het alarm te behandelen worden toegewezen in minder dan 5 seconden.

In het hierboven vermelde framework is de configuratie van query's die instaan voor de continue verwerking van de datastromen nog steeds statisch. De omgeving waarin deze query's worden uitgevoerd is daarentegen echter zeer dynamisch. Deze omgeving kan worden beschouwd als contextuele informatie in kennisgedreven systemen. In dit proefschrift wordt een onderscheid gemaakt tussen toepassingsspecifieke context en situationele context.

Toepassingsspecifieke context omvat de eerder beschreven contextinformatie zoals patiëntprofielen. Deze context kan regelmatig veranderen, en bepaalt welke individuele query's contextueel relevant zijn. Zo kan deze context de voorwaarden van de query's beïnvloeden, zoals de sensoren die moeten worden gemonitord om bepaalde

bruikbare inzichten af te leiden. Bijvoorbeeld, in de toepassing waarbij patiënten in ziekenhuiskamers worden gemonitord, dient de hoeveelheid licht en geluid in de kamer te worden gemonitord voor patiënten met een hersenschudding, omdat een te grote blootstelling aan licht en geluid een alarmerende situatie zou betekenen. Daarnaast kan toepassingsspecifieke context ook de parameters van de windows van de query's, zoals hun vereiste uitvoeringsfrequentie, beïnvloeden. Wanneer bijvoorbeeld de toestand van een patiënt verslechtert, moet de uitvoeringsfrequentie van query's worden verhoogd om de patiënt strikter op te volgen. In statische systemen vereist het beheer van deze contextgevoelige specifieke query's veel handmatige (her)configuratie door de eindgebruiker. Dit maakt dergelijke systemen zeer moeilijk te onderhouden. Generieke query's worden vaak gebruikt als alternatieve aanpak, aangezien zij minder aanpassingen vereisen. Daartoe bevatten de definities van deze query's generieke concepten uit de ontologieën, zodat de relevante sensoren worden bepaald door in real time semantisch te redeneren over alle gegevens tijdens het uitvoeren van de query. Vanwege de hoge computationele complexiteit die met dit semantisch redeneren gepaard gaat, worden zij typisch gebruikt in gecentraliseerde oplossingen. Zij vertonen dan echter weer de eerder besproken tekortkomingen.

De tweede contributie van dit proefschrift lost deze problemen op met het ontwerp van een component die adaptiviteit aan toepassingsspecifieke context integreert in een semantisch IoT-platform. Deze component, die gebouwd is met behulp van de technologieën van het semantische web, heet DIVIDE. DIVIDE kan op een automatische en adaptieve manier de query's afleiden en configureren die continu uitgevoerd dienen te worden op de semantische redeneercomponenten voor datastromen op de lokale randapparaten van het netwerk. Het zorgt ervoor dat te allen tijde enkel query's worden uitgevoerd met contextueel relevante voorwaarden en parameters van hun windows, steeds op basis van de huidige toepassingsspecifieke context. Wanneer bijvoorbeeld de diagnose van een hersenschudding wordt toegevoegd aan het medische profiel van een gehospitaliseerde patiënt, zorgt DIVIDE er automatisch voor dat de hoeveelheid licht en geluid in de kamer wordt gemonitord. Op die manier is manuele herconfiguratie van query's niet langer vereist wanneer toepassingsspecifieke context wijzigt. Door zijn ontwerp zorgt DIVIDE ervoor dat de resulterende query's alleen eenvoudige filtering vereisen en dus efficiënt kunnen worden uitgevoerd op lokale IoT-apparaten met lage specificaties. Bovendien stelt DIVIDE eindgebruikers in staat om privacy in het ontwerp van toepassingen te integreren door hen de volledige controle te geven over welke gegevensabstracties over het netwerk kunnen worden verzonden en welke gegevens de lokale omgevingen van het netwerk niet mogen verlaten.

Naast toepassingsspecifieke context omvat contextinformatie over de omgeving ook de situationele context waarin semantische query's worden uitgevoerd. Dit wordt gedefinieerd als alle externe context die beschikbaar is over de omgeving, zoals de huidige netwerkomstandigheden, de belasting van lokale apparaten, en de performantie van de semantische query's. Deze situationele context verandert voortdurend door externe invloeden. Bijgevolg vereist ook dit de nodige adaptiviteit van de query's op de semantische redeneercomponenten voor datastromen. Zo kan de optimale verdeling van query's in het IoT-netwerk bijvoorbeeld afhangen van de netwerkomstandigheden. De parameters van de windows van de query's vormen een ander voorbeeld: hun

optimale waarden kunnen afhangen van de huidige belasting van het apparaat waarop de query's worden uitgevoerd. Belangrijk is dat de optimale distributie en configuratie van query's op basis van de situationele context kan verschillen per toepassing, aangezien elke toepassing verschillende afwegingen dient te maken die een evenwicht proberen te vinden tussen performantie en andere vereisten.

Om deze uitdaging op te lossen, breidt de derde contributie van dit proefschrift het methodologische ontwerp van DIVIDE verder uit. Deze uitbreiding zorgt ervoor dat DIVIDE de verdeling van query's in het netwerk (zijnde het apparaat waarop elke query wordt uitgevoerd) en de configuratie van de parameters van de windows van de query's adaptief kan aanpassen. DIVIDE maakt dit mogelijk doordat het voortdurend de situationele context monitort op de apparaten in het IoT-netwerk die instaan voor de verwerking van de datastromen. Het ontwerp van DIVIDE bevat een ontologie die de gemonitorde context semantisch kan verrijken, evenals meta-informatie over de huidige configuratie en distributie van de query's in het netwerk. Hierdoor kunnen eindgebruikers semantische query's configureren die voor elke toepassing dynamisch bepalen hoe bepaalde parameters van de situationele context de distributie en configuratie van de query's moeten beïnvloeden. Op die manier kunnen toepassingsspecifieke afwegingen automatisch worden gebalanceerd en kan het platform op efficiënte wijze semantisch redeneren over datastromen.

De laatste contributie van dit proefschrift gaat in op de noodzaak om het framework ontworpen in dit proefschrift, dat adaptief en trapsgewijs semantisch redeneren mogelijk maakt, in te bedden in een volledig semantisch platform dat bijdraagt aan het integreren en sluiten van een feedbacklus in IoT-toepassingen. Dit betekent dat semantische services en workflows moeten worden gekoppeld aan de gebeurtenissen en inzichten die worden gegenereerd door de klassieke semantische redeneercomponenten op datastromen van een semantisch IoT-platform. In de gezondheidszorg is dit bijvoorbeeld belangrijk om acties en suggesties voor zorgverleners te ondersteunen. Om dit te realiseren presenteert de vierde en laatste contributie van dit proefschrift een referentiearchitectuur, gestoeld op de visie van trapsgewijs semantisch redeneren, met extra bouwstenen die zijn ontworpen met technologieën van het semantische web. Specifiek voor de gezondheidszorg toont deze architectuur aan hoe het mogelijk wordt om dynamische, toepassingsspecifieke, datagedreven services te instrumenteren, en om workflows te construeren met een semantische workflowcomponent, die kunnen worden gecoördineerd tussen organisaties en belanghebbenden die betrokken zijn bij de zorgverlening aan patiënten. Deze services en workflows kunnen worden gekoppeld aan de componenten van het platform die instaan voor het semantisch redeneren op de datastromen en door DIVIDE worden beheerd. Op die manier wordt de feedbacklus in het platform gesloten: de opgedane kennis kan de toepassingsspecifieke context bijwerken, waardoor DIVIDE op zijn beurt de contextgevoelige query's adaptief kan aanpassen. Binnen de gezondheidszorg kunnen de semantische componenten van het resulterende semantische zorgplatform zo tezamen worden ingezet om continue zorgtoepassingen verder te optimaliseren.

Samengevat maakt dit proefschrift adaptief semantisch redeneren op stromen van IoT-gegevens mogelijk in toepassingsgebieden van het IoT, zoals de gezondheidszorg, om zo de tekortkomingen en problemen in huidige toepassingen mee op te lossen.

Met de verschillende contributies en evaluaties op verscheidene toepassingen in de gezondheidszorg toont dit proefschrift aan hoe continue zorg op een adaptieve en automatische manier kan worden geoptimaliseerd op basis van de beschikbare gegevens. Daarbij wordt ervoor gezorgd dat het resulterende platform performant en responsief is, lokale autonomie ondersteunt, en de integratie van privacy in het ontwerp van toepassingen mogelijk maakt.

Summary

The Internet of Things (IoT) is characterized by various sensors, devices, actuators and other 'things' that are connected to the internet and continuously generate and process data in a variety of application domains. One such application domain transformed by the IoT is healthcare. By 2030, the global IoT in healthcare market size is expected to more than triple to 960 billion US dollars. The IoT is integrated in healthcare in numerous ways: patients are monitored with wearables, while their hospital rooms or smart homes can be equipped with environmental sensors, localization devices, motion detectors, devices that monitor the state of electrical appliances, and much more. All these devices constantly generate data streams of real-time raw data. IoT-enabled devices are often present that can actuate on the sensed data by, for example, closing the window blinds or controlling the heating.

The raw data generated by IoT sensors is meaningless on its own. In many IoT application domains, including healthcare, a lot of domain knowledge and contextual information relevant to the application is available. In healthcare, medical domain knowledge includes knowledge about diseases, treatments, and possible alarming situations. Context information consists of the medical profile of patients (e.g., their Electronic Health Record at the hospital or general practitioner), the description of the installed sensors across patient rooms, information about the location and availability of caregivers, and much more. The availability of these data sources offers the opportunity to integrate sensor data streams with domain knowledge and context information, to derive real-time insights about the condition and environment of the patient. This would result in context-aware applications. One could even go one step further towards actionable insights, where actions are coupled to the derived insights. For example, the closest nurse could be notified to visit the patient in case an alarming situation is detected.

Integrating IoT data with domain knowledge and context information to generate real-time insights is a challenging task. This is mainly because of two reasons: data heterogeneity and data velocity. Data heterogeneity refers to the different syntaxes, formats and representations of the data (e.g., JSON or SQL) in the data streams generated by IoT sensors and devices, domain knowledge and context information, which makes integration hard. Data velocity refers to the huge rate at which IoT data streams are continuously generated. This rate can be up to more than 100 sensor observations per second per patient in healthcare applications, highlighting the need for efficient processing techniques.

To solve these challenges, knowledge-driven systems can be built. In such systems, semantics are employed to solve the data heterogeneity challenge by semantically enriching all data and integrating it in a uniform, machine-interpretable format.

Semantic Web technologies are a set of recommended technologies that perform this enrichment with ontologies, which formally describe all concepts and their relations in a domain. Semantic reasoning techniques can then be used to derive new knowledge, such as actionable insights, from semantic data using the definitions in ontologies.

Stream reasoning is a research area that tries to tackle the data velocity challenge. It adopts semantic reasoning techniques for streaming data, by incorporating them into stream processing engines. Such engines continuously evaluate semantic queries on data streams, by placing windows of a certain size on top of the stream at a certain rate that defines the query execution frequency. In semantic reasoning, the computational complexity of the reasoning is defined by the required level of reasoning expressivity. In complex IoT domains, such as healthcare, existing domain knowledge is represented in large and complex ontologies, thus requiring highly expressive reasoning. In stream reasoning, a performance trade-off therefore exists between reasoning expressivity and data velocity: a high level of required expressivity results in a low data velocity that can be processed in real-time.

A realistic set-up of an IoT application consists of a full network of IoT devices, with different local parts. In healthcare, these local parts correspond to the local environments of the different patients. To deal with the performance trade-off in stream reasoning, existing semantic IoT platforms mainly host the high-velocity data stream processing queries on high-end back-end servers in the central parts of the network, or severely limit the complexity of the reasoning and queries that can be applied. Centralized solutions however exhibit multiple shortcomings that conflict with important requirements of applications in IoT domains like healthcare. As they send all raw sensor data streams over the network to the back-end servers, there is no flexibility in managing the privacy of the user data. Moreover, as no stream processing is done on the local devices, these local devices have no local autonomy to derive certain actionable insights and actuate on them. In addition, such solutions constantly stress the network capacity and all available server resources for the continuous forwarding and processing of all data, further limiting performance and responsiveness of the system.

To move away from centralized solutions, the concept of cascading reasoning was proposed. The vision of cascading reasoning is to construct a pipeline of semantic stream reasoners: high-velocity streams are processed by low expressivity reasoners in the beginning of the pipeline, and the subsequent reasoners can increase the expressivity of the reasoning as the data velocity decreases. This vision perfectly aligns with the vision of edge computing, where heterogeneous, low-end IoT devices in the local & edge parts of networks are involved in the data processing.

The already existing vision of cascading reasoning has not yet been realized in a generic semantic framework that is distributed over a full network, that is easily applicable to IoT domains like healthcare, and that addresses the presented shortcomings associated with centralized processing architectures. Therefore, the first contribution of this dissertation is the realization of such a framework. The framework has a generic design that allows end users to construct and configure an application-specific pipeline network of stream reasoning components in an IoT network. The design of the framework specifically focuses on two aspects: improving the overall performance

of stream reasoning on data streams for IoT applications, and introducing local autonomy by exploiting the local & edge devices of the network in the stream reasoning pipeline. This way, actionable insights can be derived from the data streams in a responsive manner. This is shown for a hospital monitoring use case, where the optimal nurse to handle an alarming situation can be assigned in less than 5 seconds.

In the presented cascading reasoning framework, the configuration of stream processing queries is still static. However, in contrast, the environment in which these queries are deployed is very dynamic. This environment can be considered as contextual information in knowledge-driven systems. In this dissertation, a distinction is made between use case context and situational context.

Use case context represents the earlier described context information, such as patient profiles. Changes to this context can regularly occur. This context determines which individual queries are contextually relevant. It can influence the conditions of queries, such as the sensors that should be monitored to derive certain actionable insights. For example, in hospital monitoring, light and sound conditions should be monitored for a patient diagnosed with a concussion, as too much exposure to light and sound would imply an alarming situation. Moreover, use case context can influence the window parameters of queries, such as their required execution frequency. For example, when a patient's condition worsens, the execution frequency of queries should be increased to monitor the patient more closely. In static solutions, managing these context-aware specific queries requires a lot of manual (re)configuration effort, and is thus infeasible to maintain. Generic queries are often used as an alternative approach, since they require fewer adaptations. To this end, they use generic ontology concepts in their definitions, such that the relevant sensors are determined through real-time semantic reasoning on all data during the query evaluation. Because of the computational complexity of the required reasoning, they are typically used in centralized solutions, which exhibit the aforementioned shortcomings.

To solve these issues, the second contribution of this dissertation is the design of a component that integrates adaptiveness to use case context in a semantic IoT platform. This component, which is built using Semantic Web technologies, is called DIVIDE. DIVIDE can automatically derive and configure the stream reasoning queries on the local & edge components of a cascading reasoning platform in an adaptive manner. It ensures that queries with contextually relevant conditions and window parameters are evaluated at all times, according to the current use case context. For example, when a concussion diagnosis is added to the medical profile of a hospitalized patient, DIVIDE automatically ensures that the light and sound conditions in the room are being monitored. This way, no manual reconfiguration of queries is required anymore whenever the use case context changes. By design, DIVIDE ensures that the resulting queries only require simple filtering and can thus be efficiently evaluated on low-end IoT devices with few resources. Moreover, DIVIDE enables privacy by design: it allows end users to integrate privacy by design into the application by leaving them in full control about which data abstractions can be sent over the network, and which data should not leave the local environments.

In addition to use case context, environmental context also includes the situational context in which semantic queries are deployed. This is defined as any external context

about the environment, such as current networking characteristics, resource usage on the device, or the performance of the semantic queries. Due to external influences, such situational context is constantly changing. This again requires adaptiveness of the stream reasoning queries. For example, the optimal distribution of queries in the IoT network can depend on the network conditions, while the optimal window parameters of a local query can depend on the available resources of the local device. Importantly, the optimal query distribution and configuration based on the situational context can differ per use case, as every use case considers different trade-offs that balance performance and other requirements.

To address this need, the third contribution of this dissertation is an extension of the methodological design of DIVIDE. This way, DIVIDE is able to update the distribution of queries in the network (i.e., their execution location), and the configuration of the window parameters of queries. DIVIDE achieves this by continuously monitoring the situational context on the processing devices in the IoT network. Its design contains an ontology model that can semantically represent the monitored context as well as meta-information about the current configuration and distribution of queries in the network. This allows end users to configure semantic queries that dynamically define for every use case how certain situational context parameters should influence the query distribution and configuration. This way, use case specific trade-offs can be automatically balanced and efficient stream reasoning can be achieved.

The final contribution of this dissertation addresses the need to embed the cascading and adaptive reasoning framework designed within this dissertation in a full semantic platform, in order to close the feedback loop in IoT applications. This means that semantic services and workflows should be coupled to the events and insights generated by the classic stream reasoning components of a semantic IoT platform. In healthcare, for example, this allows supporting actions and suggestions for healthcare workers. To achieve this, the fourth and final contribution of this dissertation presents a cascading reasoning reference architecture with other building blocks built on Semantic Web technologies. It shows, specifically for the healthcare domain, how this architecture allows instrumenting dynamic, use case specific, data-driven services, and how it allows constructing workflows with a semantic workflow engine that can be coordinated across organizations and stakeholders involved in the care provisioning of patients. These services and workflows can be coupled to the stream reasoning components of the platform that are managed by DIVIDE. This closes the feedback loop in the platform: resulting knowledge can update the use case context, which in turn triggers DIVIDE to adaptively update the context-aware queries. Hence, in healthcare, the semantic components of the resulting semantic healthcare platform can be leveraged altogether to optimize continuous care solutions.

To summarize, this dissertation enables adaptive semantic reasoning on data streams in IoT application domains like healthcare, to solve the shortcomings and issues associated to this task in the current state-of-the-art. Through its different contributions and evaluations on various healthcare use cases, it shows how continuous care can be optimized in an adaptive and automatic way based on the available data. This is achieved while ensuring that the resulting platform is performant and responsive, introduces local autonomy, and enables privacy by design.

1
Introduction

"If you think that the internet has changed your life, think again. The Internet of Things is about to change it all over again!"

– Brendan O'Brien, Co-Founder of Aria Systems

The Internet of Things, or IoT in short, is the main keyword where the research in this doctoral dissertation starts from. It was first introduced in 1999 by Kevin Ashton, a British pioneer in technology who wanted to describe a system in which physical world objects were connected to the internet by sensors [1]. Over the years, the term has become widely used to describe various scenarios in which a variety of such physical objects, including sensors, devices, actuators and other 'things', are being interconnected and exchange data to perform a plethora of tasks in various application domains, such as the smart monitoring of patients in healthcare or smart traffic control in the smart cities domain [2]. Other IoT application domains include agriculture, smart home and automation, energy, retail, logistics, and more [3]. A specific and highly relevant concept in the IoT domain is a data stream: many IoT sensors and devices constantly generate data, which creates multiple streams of real-time data. In healthcare, sensors can measure the patient's physiological parameters and environmental conditions, while sensors in smart cities might measure the traffic volumes across the city. In particular, this dissertation zooms in on the processing of such data streams in the IoT. It mainly focuses on one particular IoT application domain: healthcare.

To understand the challenges that this dissertation tries to tackle, some background information is needed in different domains. First of all, it is important to

understand how the IoT is omnipresent in healthcare, and what challenges are associated with the processing of data streams generated by IoT sensors and devices. This is explained in Section 1.1. To this end, the concept of knowledge-driven approaches is introduced in Section 1.2, where semantics come into play. The research domain of stream reasoning is discussed, which exhibits an important performance trade-off associated to real-time data processing. The vision of cascading reasoning is put forward in Section 1.3 as a means to move away from centralized IoT processing architectures and their associated shortcomings, and to exploit the opportunities of local and edge device processing. Finally, before the research challenges and contributions of this dissertation are presented in Section 1.5 and 1.6, the need is identified in Section 1.4 to make stream reasoning adaptive to a constantly changing environmental context. To conclude this introductory chapter, Section 1.7 discusses the outline of this dissertation, and Section 1.8 presents an overview of all publications that were realized during this PhD study.

1.1 Healthcare and the Internet of Things

In recent years, the rise of the IoT has transformed many application domains that involve human-machine interaction. According to Transforma Insights, 29.42 billion IoT devices will be connected worldwide by 2030, as shown in Figure 1.1 [4]. This is almost double the number of today. This proves its importance in the aforementioned application domains today, and illustrates its projected further increase of importance towards the upcoming years.

Zooming in on the healthcare domain, the immense impact of the IoT cannot be underestimated [5]. While the current size of the worldwide IoT in healthcare market is estimated at 261.69 billion US dollars, it is predicted to more than triple over the next few years. As shown in Figure 1.2, by 2030, the size of this market is estimated to be at 960.2 billion US dollars [6].

The IoT is integrated in healthcare applications in various ways [5]. Patient rooms of hospitals, nursing homes and service flats in smart home environments are being equipped with sensors that monitor environmental conditions, such as light, sound and temperature level in the room. Localization devices can measure the indoor location of people across rooms through technologies, such as Bluetooth Low Energy (BLE), while movement across rooms is picked up by motion sensors. Moreover, the state of windows, doors, appliances and other IoT-enabled devices can be monitored. Wearable devices can be used to monitor physiological parameters of patients, such as heart rate and heart rate zones, as well as movement-related parameters, such as acceleration, to get insights into activity patterns [7]. In addition, the IoT is not only about sensing and collecting data, but also about actuating on the sensed data. Using IoT-enabled devices, lights, windows, blinds, heating installations and many others can be controlled and automated.

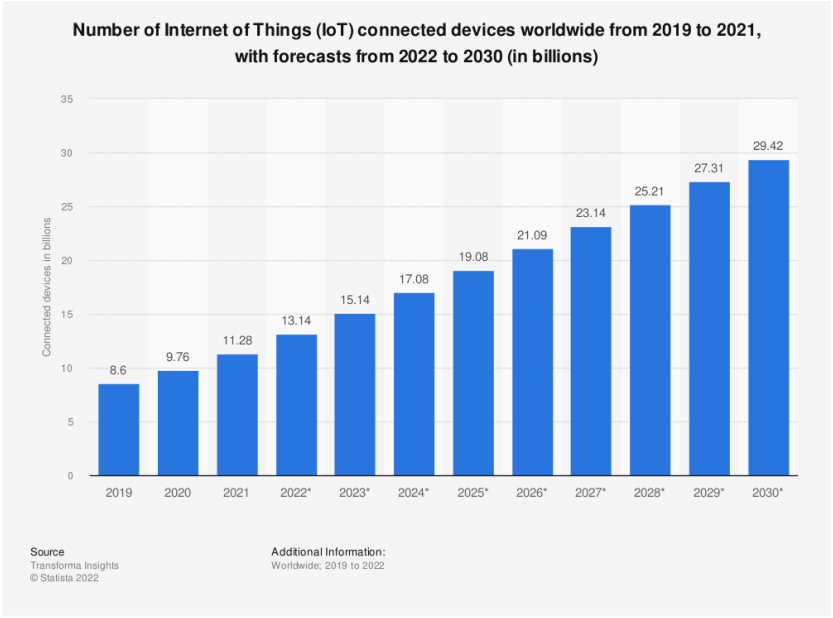


Figure 1.1: Visual overview of the report and forecast of the number of IoT connected devices worldwide [4]

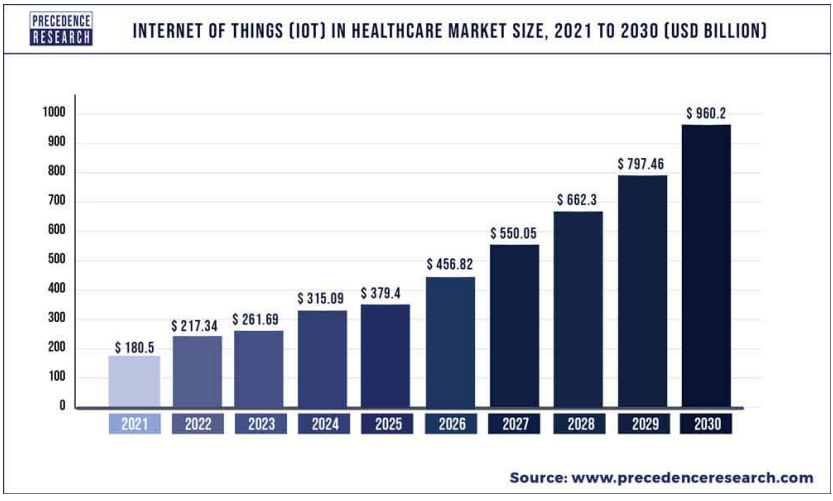


Figure 1.2: Visual overview of the report and forecast of the global IoT in healthcare market size [6]

In many IoT application domains, lots of contextual information and domain knowledge exists [8]. This is also true in healthcare [9]. Considering the example of a hospitalized patient in a care room, the Electronic Health Record (EHR) of this patient contains information about diagnoses, treatment and medical history of the patient. Medical domain knowledge includes important knowledge about diseases, associated medical symptoms and possible alarming situations. Context information about the layout of the hospital buildings in relation to the patient's room and the location and availability of the care staff can be relevant to assign a care staff member to handle alarming situations. In addition, context information includes details about the different IoT devices and sensors installed in rooms.

The available IoT sensors and devices in IoT applications constantly generate data, essentially forming large streams of data. Given the available domain knowledge and context information, the IoT offers the opportunity to integrate this information with the generated data streams [8]. If this would be possible in a uniform and efficient way, this would allow generating new knowledge and relevant real-time insights, e.g., about the patient's condition and environment in healthcare, resulting in context-aware applications. In the given example, this would allow detecting alarming situations and actuating on them by selecting the optimal care staff member to handle it, while considering the priority and context of the alarm.

Integrating the IoT in such a way into healthcare solutions and leveraging its opportunities supports the creation of Ambient Assisted Living (AAL) environments [10–12]. AAL represents a multidisciplinary field that provides solutions to individuals to improve their quality of life in various ways [13]. To this end, it connects technology with various other fields, such as sociology [14]. Enhanced Living Environments (ELE) is another umbrella term used to refer to developments in Information and Communication Technology (ICT) that support the creation of AAL environments [15]. Hence, ELE also encompasses the technological achievements and possibilities associated with the rise of the IoT in healthcare.

Integrating IoT data with domain knowledge and context information to generate real-time insights is a challenging task. This is true because of multiple reasons, of which the main ones that will be further addressed in this dissertation are:

- Data heterogeneity: IoT networks are characterized by a large heterogeneity of the different devices and sensors in terms of hardware, operating systems, and used technologies [16, 17]. The data generated by them is heterogeneous as well. It can be represented in different formats and encodings (e.g., JSON, plain text or SQL) and its quality can vary over time (e.g., the accuracy of a wearable sensor can be disturbed due to much movement of the person wearing it) [8]. In addition, the available sources of domain knowledge and contextual information can be heterogeneous as well (e.g., EHR versus a flow chart about how patients with certain diagnoses should be treated).

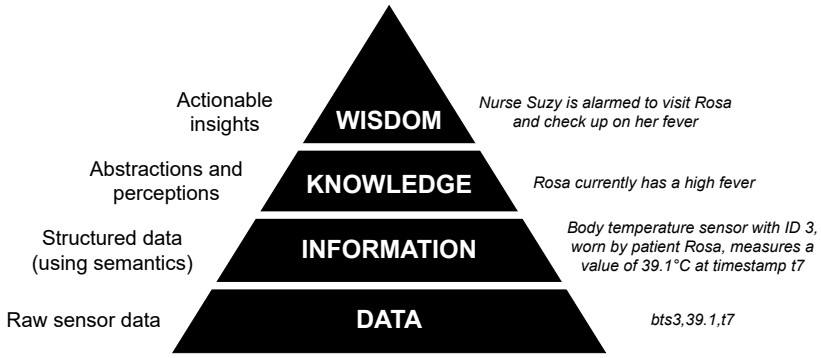


Figure 1.3: Knowledge hierarchy in the context of the IoT and semantics [8]. To the left of the triangle, the concepts from the hierarchy are mapped to the domain of semantics for the IoT. To the right of the triangle, this is illustrated with a simple example.

- Data velocity: IoT networks consist of multiple devices and sensors that continuously generate data. Every such entity produces a stream of data at its own rate. As already addressed in the examples, a wide range of data-generating entities is often deployed in applications of IoT domains such as healthcare. If you consider a full IoT network in a real-world set-up, it becomes clear that all these devices and sensors *together* generate a large amount of data. Hence, the combined rate or velocity at which IoT data is generated can become huge [18–21]. Software components need to efficiently process these high-velocity data streams to be able to generate real-time insights.

Knowledge-driven systems lie at the core of how these challenges can be solved. More specifically, semantics and stream reasoning techniques can be employed to deal with the heterogeneity and velocity of the data, respectively. These concepts are detailed in the next section.

1.2 Towards a knowledge-driven approach with semantics and stream reasoning

The previous section has introduced the concept of integrating IoT sensor data streams with domain knowledge and contextual information. This transformation process can be illustrated by considering the famous *knowledge hierarchy*, as shown in Figure 1.3, which represents the conversion of data to information to knowledge to wisdom [22]. The interpretation of these four layers can be adapted to the specific context of the IoT [8].

In the IoT, the data layer of the knowledge hierarchy is represented by the raw sensor data. This raw sensor data is meaningless on its own. To become information,

it has to be integrated with available context information. In other words, the correlations between the context data & measurements and between the measurements themselves need to be made explicit. This is possible by semantically enriching the data and integrating it using a uniform, machine-interpretable format. As such, the meaning and context of the data becomes clear [23]. This way, raw data, such as *bts3,39.1,t7*, can be semantically enriched to describe that *bts3* is the ID of a body temperature sensor, which observes a value of 39.1 degrees Celsius (°C) at timestamp *t7*. Other context information can then be used to link the body temperature sensor with ID *bts3* to a patient Rosa, who is hospitalized in a certain room and has a certain medical profile.

Semantic Web technologies are a set of technologies recommended by the World Wide Web Consortium (W3C)[1] that allow semantically enriching and integrating heterogeneous data sources by using ontologies. An ontology is a semantic model that formally describes concepts in a particular domain, their relationships and attributes [24]. Figure 1.4 illustrates how an ontology can be used to semantically annotate sensor observations and associated context information. It shows how different concepts can be represented by ontology classes, and how object properties can be used to describe the relationships between classes. For example, the object property `madeBySensor` describes the relationship between the classes `Observation` and `Sensor`. Data properties are used to connect classes to literals of a certain datatype. For example, the data property `hasValue` can connect the class `Observation` to a literal of datatype `xsd:double`. Instantiations of a certain class are called individuals. The figure illustrates how the body temperature observation in Figure 1.3 can be modeled using a collection of individuals and literals. For example, `obs373` is the individual that is instantiated from the class `BodyTemperatureObservation`. It is linked to the literal `°C` via the data property `hasUnit`. The additional context that links the given observation to the patient Rosa, is also shown in the figure.

In the set of Semantic Web technologies, the Web Ontology Language (OWL) is the W3C standard to define ontologies [25]. The Resource Description Framework (RDF) allows representing different ontology concepts and their relations as a directed graph of triples [26]. A triple consists of a subject, a predicate and an object. In Figure 1.4, the bottom right box includes a representation of the sensor observation and context information in RDF/Turtle, which is a compact serialization format for RDF data that makes use of prefixes. An example of a triple in this figure in RDF/Turtle is `:Rosa rdf:type :Patient`, where `:Rosa` is the subject, `rdf:type` is the predicate, and `:Patient` is the object. The SPARQL Protocol and RDF Query Language (SPARQL) can be used to write and evaluate queries on RDF data [27].

Semantic reasoning is a technique to interpret semantic data and derive new knowledge from it using a set of axioms defined in ontologies. It is required in the knowledge hierarchy of Figure 1.3 to go from information to knowledge. By describing semantic definitions in ontologies, semantically annotated sensor events can be

[1] Website: https://www.w3.org/

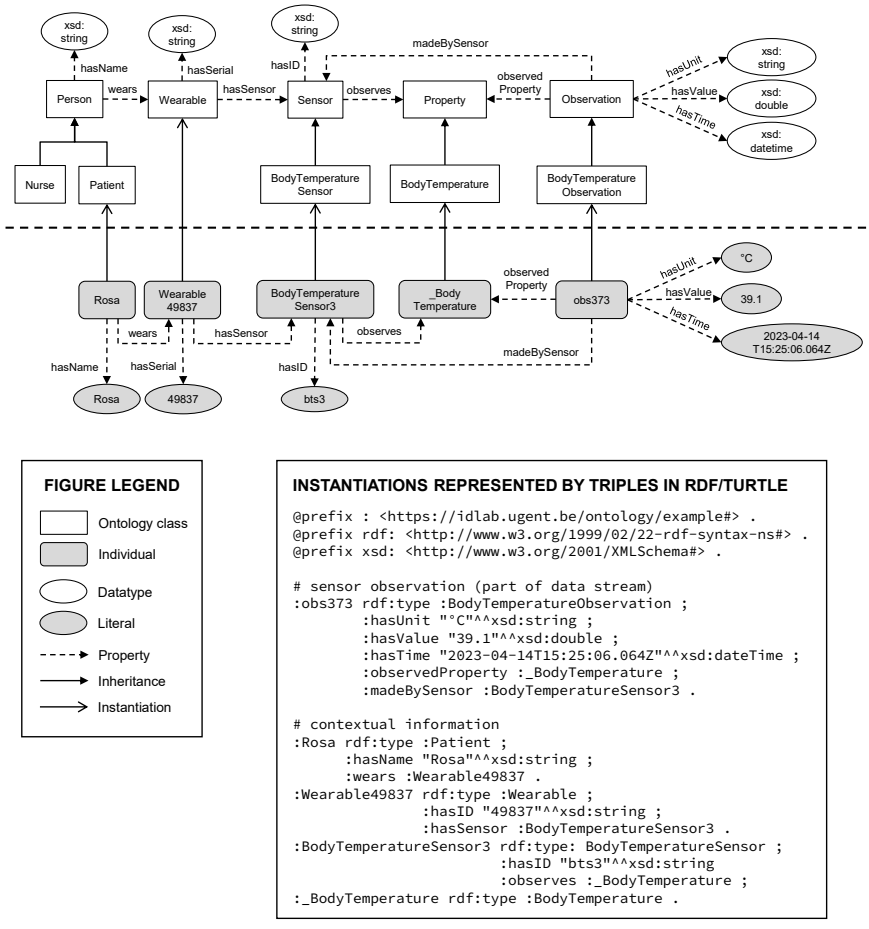


Figure 1.4: Illustration of a simple ontology for a healthcare use case and how it can be used to semantically represent a body temperature sensor observation and associated contextual information. The upper part of the figure above the dashed line represents the ontology concepts and their relations, while the entities with a gray background below the dashed line represent instantiations of those concepts. The represented sensor observation corresponds to the example in Figure 1.3. The bottom right box contains an RDF representation of all shown instantiations in the RDF/Turtle format.

further abstracted. For example, a body temperature observation above 39°C could be abstracted as a high fever event. This way, a semantic reasoner could derive that Rosa's body temperature observation of 39.1°C in the example of Figures 1.3 and 1.4 is such a high fever event, using the following definition:

```
HighFeverObservation ≡ BodyTemperatureObservation
    and hasValue some xsd:double[> "39.0"^^xsd:double]
```

The IoT allows even further transforming this knowledge into actionable insights: if a high fever event is detected, a semantic reasoner could derive that the closest nurse, Suzy in the example of Figure 1.3, should be notified to visit Rosa's room and check up on her fever.

To summarize, the previous paragraphs have shown how knowledge-driven systems can be built. By using semantics to integrate sensor data streams with domain knowledge and context information and applying semantic reasoning, actionable insights can be derived.

The presented concepts allow dealing with the data heterogeneity challenge presented in the previous section. However, they do not yet specifically tackle the data velocity challenge, since classic semantic reasoning techniques are not designed to support data streaming use cases. Zooming in on semantic reasoning algorithms, it should be noted that their computational complexity depends on the expressivity of the underlying ontology [28]. Different OWL sublanguages exist, which all vary in degree of expressivity [29]. They range from RDFS, which has the lowest expressivity, to OWL 2 DL, which supports highly expressive reasoning with much more complex relations. In complex IoT domains, such as healthcare, ontologies representing existing domain knowledge are often large and complex, and thus require more expressive semantic reasoning [30]. For example, the well-known Systematized Nomenclature of Medicine Clinical Terms (SNOMED-CT) ontology contains more than 350,000 concepts that are organized in complex hierarchies [31, 32]. As a consequence, semantic reasoning on such ontologies is slow and requires a lot of computational resources.

To tackle the data velocity challenge, stream reasoning (SR) has emerged as a challenging research area that mostly focuses on the adoption of semantic reasoning techniques for streaming data, by trying to incorporate them in stream processing engines [21, 33]. In general, stream processing engines continuously process data streams that originate from different sources, to produce new data streams as output of the engine [34]. By combining this concept with the adoption of semantics through ontologies and semantic reasoning, stream reasoning allows addressing both the data heterogeneity and data velocity challenges in the IoT. This is summarized in Figure 1.5.

RDF Stream Processing (RSP) is a subdomain of stream reasoning that focuses on the integration of RDF data streams with context information, and the continuous evaluation of RSP queries on this integrated data model while semantic reasoning is performed to a certain extent [21]. Different RSP engines exist, such as C-SPARQL [35] and SPARQL$_{Stream}$ [36]. Because a data stream has no beginning

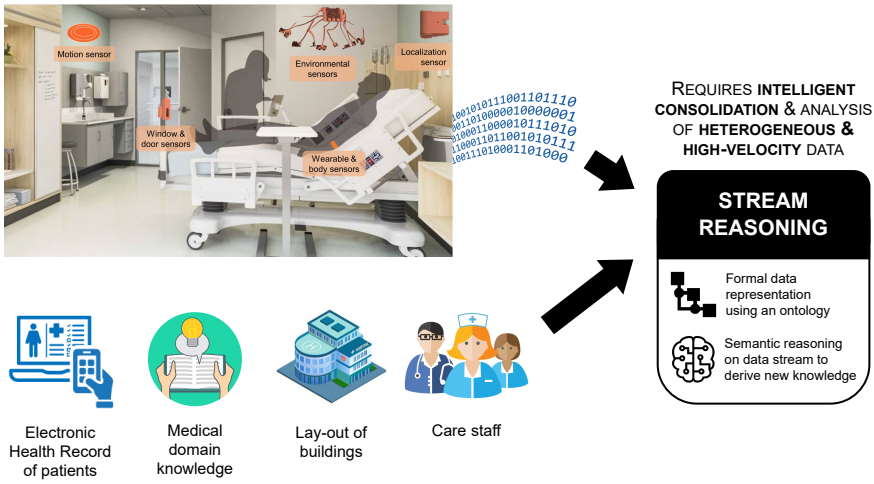


Figure 1.5: Schematic overview of using stream reasoning, for the healthcare example of a hospital monitoring use case. The figure visualizes how stream reasoning can consolidate and analyze data from various sources: data streams of IoT sensor data on the one hand, and domain knowledge & contextual information about patient & environment on the other hand.

or ending, most RSP engines place a window on top of it. Continuous queries are then registered once and produce results continuously over time as the streaming data passes through the window. This process is visually illustrated in Figure 1.6. The input data windows of RSP queries are defined through window parameters: the size of the data window and the sliding step. The latter defines the size of the steps between every window placed on the data stream, and thus directly implies the query execution frequency.

Throughout this dissertation, the term 'stream processing' will be often used. When this term is used within the context of stream reasoning or stream processing components of a semantic IoT platform, note that this implicitly refers to RDF stream processing.

Most RSP engines only support semantic reasoning of low expressivity during the query evaluation, as the velocity of the data is too high to perform more expressive semantic reasoning in a responsive manner [33]. In other words, a performance trade-off exists within the area of stream reasoning between required reasoning expressivity and typical data velocity. The required level of real-time processing and thus responsiveness can vary per use case, but in many use cases of IoT application domains such as healthcare, making certain decisions is time-critical. For example, in hospital care, generating insights on the conditions of the patient and environment cannot exhibit a high delay, as alarming situations should be handled within 5 minutes in various countries.

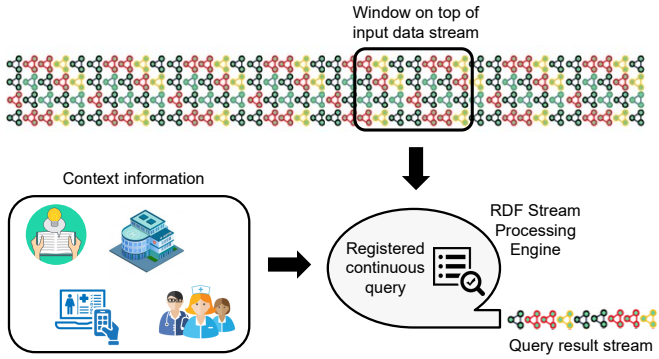


Figure 1.6: Schematic overview of the evaluation of continuous queries in RDF Stream Processing engines. Continuous queries are registered to the engine, and are continuously evaluated over time as the streaming data passes through the window. The query is evaluated on a data model that integrates the streaming data window with context information (and possibly domain knowledge). This continuous query evaluation generates a continuous stream of query results.

In this context, it is important to note that the usage of the term 'real-time' in this dissertation does not refer to *hard* real-time or *soft* real-time. Instead, in this dissertation, 'real-time' should be considered as *near* real-time, which imposes a softer restriction than *hard* real-time and leaves room to individual use cases to analyze what is considered as the required level of responsiveness.

1.3 Moving away from centralized processing with cascading reasoning

Considering the examples of typical applications in an IoT domain like healthcare, such as homecare monitoring and hospital monitoring, a realistic real-world set-up consists of an actual IoT *network* of sensors and devices, with different local parts corresponding to the local environments, e.g., of the different patients. This is depicted in Figure 1.7 for a healthcare application. In such an IoT network, there is a central server environment or cloud environment, which hosts the central servers of for example a nursing home or hospital. This central environment also contains the domain knowledge and the relevant context information. The latter includes for example the EHRs of all patients in the system. Moreover, the multiple patients are spread out over the IoT network. For every such patient, there is a monitored environment equipped with sensors and devices. These parts of the IoT network are considered the local and edge parts, as opposed to the central server side. Depending on the considered application, these local parts can be spread out over the entire city, hospital environment, nursing home, or a mix of locations.

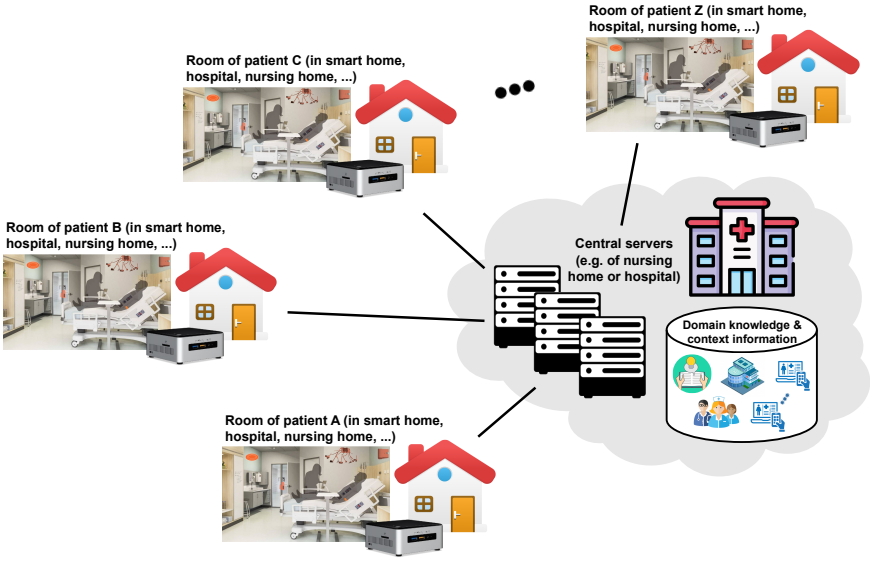


Figure 1.7: Visual overview of a typical IoT network in a healthcare application. It consists of a central server or cloud environment hosting central servers and the available knowledge & context information, and a local part for every patient with monitoring sensors and devices.

To deal with the performance issues associated with the high computational complexity of semantic stream reasoning on high-velocity data streams, which was discussed in the previous section, classic solutions are often centralized. In Figure 1.7, this corresponds to a situation where the central servers do all the actual processing of the sensor data streams generated in the system, for all existing patients. Following this approach, the high-end servers with most resources can then perform the data stream processing. To this end, the local environments each contain a local hub that combines the data from all sensors in the local environment and forwards it to the cloud. Such centralized IoT processing architectures exhibit multiple shortcomings that conflict with the other, non-performance related requirements of many IoT applications in general and healthcare applications in particular. These shortcomings are summarized in Figure 1.8.

First, centralized solutions process all the raw sensor data streams on the back-end servers. This implies that all raw sensor data is sent unfiltered to the back-end over the backbone network. This way, such solutions do not offer any flexibility in managing the privacy of the generated data, for example by keeping data local or abstracting the information that is sent over the network. This is undesirable, as privacy management is an important consideration in IoT domains such as healthcare [37]. Therefore, a semantic IoT platform should enable privacy by design [38]: it should allow an end

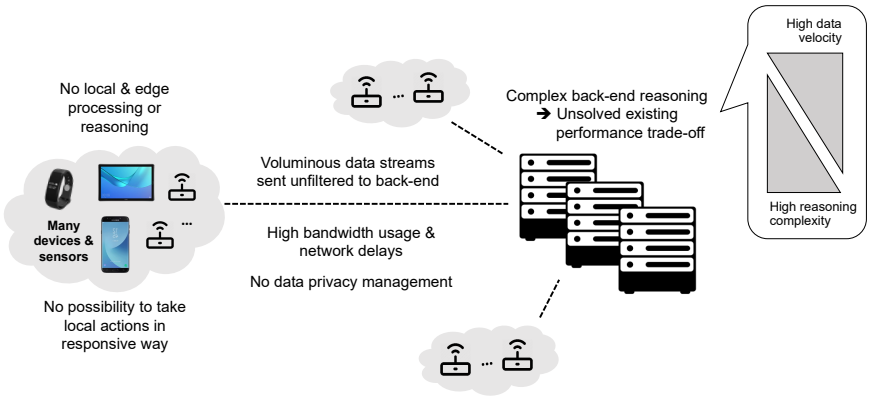


Figure 1.8: Overview of the shortcomings of centralized IoT architectures that perform back-end only reasoning on the data streams generated in IoT applications

user to build privacy by design into an application by precisely defining, on different levels of abstraction, which data is kept locally and which data is sent over the network.

Second, processing all data server-side implies that no data processing is performed locally. Since multiple IoT devices are typically present in the local environments of the network, doing some of the data processing locally would give the solution some local autonomy: certain actionable insights could be derived locally. This is useful in certain IoT applications to actuate on certain situations in a responsive manner, for example in healthcare by alerting a nurse in an alarming situation or updating the environment (e.g., dimming the lights) according to the condition of the patient. However, in centralized solutions, there is no such local autonomy.

Third, sending the raw, unfiltered high-velocity sensor streams to central servers also constantly stresses the network by using a lot of bandwidth, possibly incurring high delays as well. This can further reduce responsiveness of the system, which has an additional negative impact on the issue of having no local autonomy as well.

Fourth, given the performance issues associated to the processing of high-velocity data streams, high-end hardware is required server-side to do this for *all* data streams (i.e., for all patients in a healthcare application) in the network. Even if the budget is available to support the incurred costs, this would imply that the server resources are constantly stressed with a high load, which is highly undesirable.

To move away from centralized solutions, the vision of cascading reasoning was proposed [28], as illustrated in Figure 1.9. A pipeline of semantic reasoners is defined where the beginning of the pipeline uses low expressivity reasoning on the high-velocity data streams. The further in the pipeline, the lower the velocity of the data stream gets, and thus the higher the expressivity of the reasoning can become.

The vision of cascading reasoning in stream reasoning can be directly mapped to the general vision of fog computing and edge computing, where the processing of data

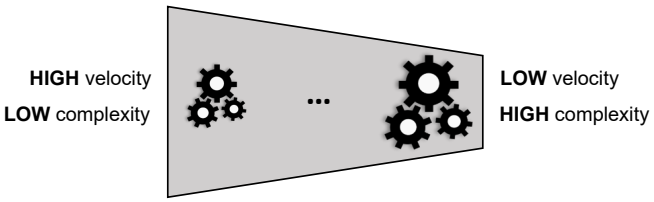


Figure 1.9: Simplified schematic overview of the concept of cascading reasoning. The concept defines a pipeline of reasoners where high-velocity streams are processed by low expressivity reasoning with low complexity in the beginning of the pipeline, and the resulting low-velocity data streams are processed by high expressivity reasoning of high complexity at the end of the pipeline.

in IoT networks starts further away from the central components [39, 40]. It introduces one or more additional layers in the data processing pipeline between the data acquisition and cloud-based processing layer, where data can be filtered before it is forwarded to the cloud. In the context of cascading reasoning, this would map the first stream reasoning components in the pipeline to the existing local & edge devices in the IoT network, i.e., in the patient's environment in healthcare. These are often low-end devices with fewer resources, making it even more important to properly realize the vision of cascading reasoning as a solution to the aforementioned performance trade-off.

Figure 1.10 illustrates the concept of cascading reasoning with a healthcare example of a cascading reasoning pipeline in an IoT network. In this example, two stream reasoning components are connected. The first component is hosted in every patient's local environment and processes the high-velocity, raw accelerometer data generated by the patient's wearable. Through continuous queries, activity patterns are detected from the accelerometer data. These activity patterns form a stream of lower velocity and are forwarded over the network to the second stream reasoning component. This component is hosted on the servers in the cloud environment and performs anomaly detection on the different streams of incoming activity patterns of all patients in the network, to detect any anomalous behavior of the patients. The output of this component is again a stream of lower velocity with the anomalies of all patients. This stream can be forwarded to other cloud components that can act on the generated knowledge.

The vision of cascading reasoning has not yet been fully realized in a generic semantic framework that is easily applicable to healthcare or other IoT application domains, addressing the presented shortcomings associated with centralized processing architectures. Existing frameworks in the domains of AAL and ELE do not combine the principles of semantic stream reasoning, cascading reasoning and edge computing. The realization of such a generic cascading reasoning framework would fit within the generic reference architecture for AAL and ELE platforms [13], which is essentially based on cascading and edge computing principles as well.

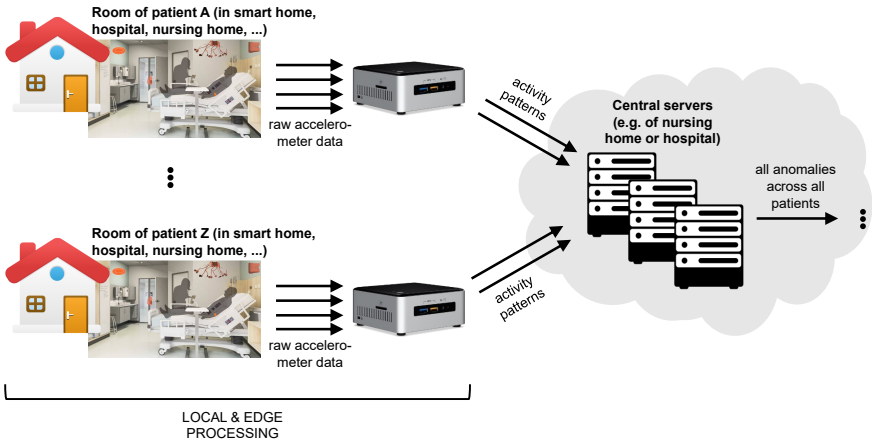


Figure 1.10: Illustration of the concept of cascading reasoning with a healthcare example of a cascading reasoning pipeline in an IoT network. In the example, two stream reasoning components are chained: one local component per patient processing the high-velocity accelerometer data streams and generating activity patterns, and one central component detecting anomalies from the activity patterns of all patients.

1.4 The need for making stream reasoning adaptive to constantly changing environmental context

The previous sections have detailed how the cascading reasoning vision has the potential to balance the existing performance trade-off in semantic stream reasoning *and* take into account other requirements associated to general IoT and healthcare applications, *if* it were properly realized in a framework that fits with the principles of fog & edge computing. Such a generic framework could then be employed in specific semantic IoT platforms to offer solutions in various IoT applications, e.g., in healthcare use cases. Semantic IoT platforms are an umbrella term for platforms that consist of different semantic components deployed across the IoT network, to perform the processing of data. There is however still another important requirement that remains unaddressed up to now: the need for stream reasoning to become adaptive to the constantly changing environmental context. This requirement needs to be taken into account as well in generic cascading reasoning architectures, for various reasons. These reasons are highlighted in this section.

The need for adaptiveness follows from the dynamic nature of the environment in which tasks are deployed in a typical IoT network. This is definitely the case for healthcare applications as well. As explained before, tasks in semantic stream reasoning platforms are represented by continuously evaluated queries. Following the approach of semantics, the constantly changing environment can be considered as contextual information. This environmental context consists of two distinct parts:

use case context, and situational context. These terms will be used in this dissertation, with their definitions according to the following paragraphs.

Use case context is any contextual information that is directly linked to the use case at hand. This is the context that was mainly described in the first section of this chapter, such as the EHR or medical profile of a patient in healthcare. This context offers the possibility to make both the conditions *and* window parameters of semantic queries context-aware. In terms of query conditions, the context determines which sensors and devices should be monitored by queries (and which other sensors and devices can thus be ignored), and what (actionable) insights can be generated from the data. In a hospital monitoring use case, the patient's diagnosis in the EHR determines the monitoring tasks that should be performed, and what events should be considered as alarming situations. For example, when a patient is diagnosed with a concussion, the light and sound conditions should be monitored, since too much exposure to light and sound would imply an alarming situation. In a homecare monitoring use case, the location of a patient in a service flat determines which in-home activities can be monitored, and which sensors should be used for that. For example, in a bathroom, the humidity should be monitored to check when the patient is showering, while monitoring this property is less relevant when the patient is in another room, such as the living room. Considering the window parameters of queries, real-time conditions of a patient might influence the frequency at which certain monitoring queries should be executed. For example, if a patient's condition worsens, the sliding step of the query window may need to be decreased, in order for alarming situations to be detected earlier and to thus more closely monitor the patient.

In general, changes to the use case context are not infrequent. For example, in healthcare, details of a patient's diagnosis or treatment may be updated in the EHR, while the in-home location of a patient in a service flat obviously changes over time as well. Therefore, to make the queries on the stream processing components of semantic IoT platforms context-aware at all times, these changes should be taken into account when configuring the queries' conditions and window parameters. In existing semantic IoT platforms, the configuration of these queries is however not adaptive and automated, but still a manual task. This means that it is practically infeasible to work with specific stream processing queries that filter the contextually relevant sensors for one specific task. If this approach would be chosen, the queries should be manually updated whenever the context changes. Since this is infeasible in practice, current platforms mostly work with generic queries instead. Such queries use generic ontology concepts in their definitions, so that semantic reasoners can reason in real-time on all sensor data, domain knowledge and context information to determine the sensors and devices to which the query is applicable. To illustrate the difference between a specific query and a generic query, consider the example query in healthcare that is responsible for monitoring the sound level in a room of a patient with

a concussion. This task could be performed by both a specific and a generic query, which could be textually defined as follows:

- Specific query: generate an alarming situation *if* the sensor with ID 126 in the room with number 7 observes a value higher than 30 decibels

- Generic query: generate an alarming situation *if* all the following conditions are fulfilled: (i) a sensor with ID A in a certain room B is observing a certain property C; (ii) a patient D is hospitalized in room B; (iii) patient D has a diagnosis that implies, according to medical domain knowledge, that the property C should not exceed the threshold value E; (iv) the sensor with ID A observes a value higher than threshold E

Generic queries do not need to be updated that often when the use case context changes, but are computationally intensive because of the involved semantic reasoning. Hence, to avoid the aforementioned performance issues, current solutions typically evaluate these queries on central components. As explained in the previous subsection, multiple shortcomings are associated with such a centralized approach. To conclude, it follows from these observations that there is a need for a component that can make the conditions and window parameters of stream processing queries adaptive to changing use case context, in an automated way.

In addition to use case context, there is also situational context. Situational context in which semantic queries are deployed can be defined as any external context information about the environment. Examples include current networking characteristics, resource usage on the device, the performance of the semantic queries, the properties of the data streams, etc. Due to external influences that are out of control of the stream processing engines, this context constantly changes. For example, the conditions of a full IoT network deployed across a city can vary over time (e.g., in terms of available bandwidth), while the low availability of resources on the low-end local processing devices could have a big impact on the performance of simultaneous processes. Similarly to changing use case context, such changing situational context requires adaptiveness of the deployed queries. More specifically, it might influence the optimal distribution of the queries in the IoT network across processing devices, and the configured query window parameters. For example, the available network capacity might influence whether it is practically feasible to send over all raw data streams to the central server to do the processing or not, while the query performance on low-end devices might require a decrease of the query frequency if the in-memory data processing cannot keep up with the data velocity. Considering this need for adaptiveness, it should also be noted that the optimal query configuration and distribution based on situational context ideally is use case specific as well, as every use case needs to consider different trade-offs that balance performance and other use case specific requirements. For example, some use cases might prefer some processing queries to be performed

locally as much as possible because of privacy constraints, while other use cases might require central processing in order to visualize raw data in user dashboards.

To summarize, when realizing the vision of cascading reasoning, it is important that the stream reasoning tasks (queries) deployed across the network in semantic IoT platforms are adaptive to a constantly changing environmental context. This includes the query conditions and window parameters based on the use case context, as well as the window parameters and query distribution according to the situational context.

1.5 Research challenges

From the previous sections, it follows that multiple challenges exist in the research field of semantic stream reasoning on IoT data streams for IoT application domains such as healthcare. This section collects them into four specific research challenges that constitute the problem statement of this doctoral dissertation, and will thus be focused on in its contributions.

Research challenge RCH1: Performant & responsive real-time stream reasoning with local autonomy across a heterogeneous IoT network

Stream reasoning is challenging in complex IoT domains, such as healthcare, since ontologies represent complex domain knowledge and thus require highly expressive semantic reasoning to derive new (actionable) insights. This is especially true in use cases where real-time insights and actions need to be derived in a responsive manner. In existing centralized solutions, all data streams are sent unfiltered to the central servers to perform the stream processing. These solutions cannot efficiently perform real-time stream reasoning tasks for all data streams across a full IoT network, as they do not solve the trade-off between reasoning expressivity and data velocity. Moreover, they do not offer any local autonomy to responsively react to certain situations. This was shown in Figure 1.8. The concept of cascading reasoning (Figure 1.9) has the potential to balance the performance trade-off and integrate local autonomy by including the heterogeneous devices in the local & edge parts of the network as processing devices. Such devices are typically low-end devices with limited resources, increasing the performance constraints of the local processing. However, the vision of cascading reasoning has not yet been fully realized in a generic semantic framework that is easily applicable to IoT applications.

Research challenge RCH2: Adaptive configuration of stream processing queries based on use case context, enabling privacy by design

In IoT applications, context-aware queries need to be evaluated on the stream processing components of semantic IoT platforms. Existing platforms cannot dynamically adapt those queries to changing use case context in an automated way. This is

true for both the conditions of those queries and their window parameters. Therefore, configuring and managing those specific, context-aware queries is a manual task that requires a lot of effort from the end user. Changes in use case context require a manual reconfiguration, which is infeasible to maintain. Existing centralized solutions tackle this issue by using generic queries that reason on all generated sensor data and available domain knowledge & context information, which reduces the frequency at which a manual reconfiguration is required. However, this centralized approach does not take the privacy of user data into account, as it sends all data to the central servers by default. This way, this approach does not enable privacy design by letting the end user in control about which data is kept locally and which data (abstractions) can be sent over the network. This is also shown in Figure 1.8. Hence, there is a need for a solution that automatically adapts the stream processing queries based on use case context, and enables privacy by design.

Research challenge RCH3: Adaptive configuration and distribution of stream processing queries based on situational context

Existing stream reasoning platforms cannot dynamically adapt to changing situational context. The configuration and distribution of queries across the components in an IoT platform often balances use case specific trade-offs. This query configuration and distribution should not only be updated when the use case context changes, but also when the situational context changes. In dynamic environments, this context constantly changes at an even higher rate than the use case context. Therefore, an ideal solution would continuously monitor the situational context and use this to adaptively update the distribution of queries across the network and the configuration of their window parameters. However, there is no one-size-fits-all solution as to how this context should influence the query configuration and distribution: different use cases have different requirements, and need to balance different trade-offs. Hence, the influence of situational context on the query distribution and configuration should be configurable by an end user, while the adaptation itself should be automated.

Research challenge RCH4: Closing the loop by embedding the solutions into a full semantic platform that is efficient & performant

It should be possible to embed the solutions to the previous challenges into a full semantic platform, in order to close the feedback loop in IoT applications. This means that semantic services and workflows should be coupled to the events, abstractions and insights generated by the stream processing & stream reasoning components of the semantic IoT platform in an efficient and performant manner. These could then directly link actions to the monitored information, which should trigger the actuators. The resulting services, workflows and actuators should then update the actual use case context, which would then at its turn adaptively update the context-aware stream processing queries. This way, the feedback loop would become closed.

1.6 Research contributions & hypotheses

The overall objective of this doctoral dissertation is to enable adaptive and performant semantic reasoning on data streams in IoT applications, with a focus on the healthcare application domain. To achieve this objective, the research challenges outlined in Section 1.5 are tackled in several research contributions. For each contribution, one or multiple hypotheses are formulated that are validated in the dissertation. These contributions and hypotheses are discussed in this section, followed by an overview of the healthcare use cases considered for the evaluations in this dissertation. The coherence of the different contributions is visually shown in Figure 1.11.

Research contribution RC01 – A generic cascading reasoning framework enabling performant & responsive real-time reasoning with local autonomy across a heterogeneous network

The first contribution of this dissertation addresses research challenge RCH1. It realizes the vision of cascading reasoning in a responsive manner, allowing for local autonomy. This is achieved by designing the architecture of a cascading reasoning framework built on Semantic Web technologies, that takes into account the principles of fog & edge computing by exploiting the full network topology to perform processing tasks. It has a generic design that can be mapped to a semantic IoT platform: it offers the tools to enable stream reasoning for the IoT by constructing and configuring an application-specific pipeline network of stream processing components across an IoT network. As streaming data flows through the pipeline, the data volume and velocity decreases while the expressivity of the semantic reasoning increases. This makes it possible to easily apply the framework to several use cases in IoT application domains that require the responsive real-time processing of streaming data. By moving away from centralized processing architectures and involving the heterogeneous, low-end local & edge devices of the IoT network, local autonomy can be achieved.

This research contribution will investigate the following hypotheses:

Research hypothesis RH1: The realization of a generic cascading reasoning framework in an IoT network will improve the overall performance of semantic stream reasoning on IoT data streams. The full pipeline of stream reasoning components will be able to generate relevant actionable insights from events in the data and handle those events in less than 5 seconds.

Research hypothesis RH2: The realization of a generic cascading reasoning framework in an IoT network will introduce local autonomy by letting local & edge devices in the network host queries. This will allow certain events in the data to be handled locally through actionable insights derived from the data, without requiring human intervention or involving central reasoning components. The local & edge components in the pipeline will also be able to perform these tasks in less than 5 seconds.

The architecture of the designed cascading reasoning framework fits within the generic reference architecture for AAL and ELE environments. This framework is the first in the domain of AAL to combine real-time and expressive stream reasoning in a cascading fashion, taking into account the principles of fog & edge computing as well.

Considering the research hypotheses RH1 and RH2, an example in healthcare of relevant events that should be handled within 5 seconds are alarming situations in hospital care. To correctly handle those, necessary actionable insights that should be derived include possible local actions that could be taken to solve the alarming situation, and generating a nurse call and selecting an appropriate nurse to come to the hospital room in case additional (human) intervention is needed to address the alarming situation. In hospital care, various countries demand that every alarm is handled by a nurse within 5 minutes after the alarm. To reduce the impact of human factors as much as possible, this requires that every nurse call assignment is completed within 5 seconds after the alarming situation first begins. This restriction leaves ample time for the nurse to move to the correct location after receiving the nurse call alert. In other IoT application domains, the threshold of 5 seconds is also relevant. For example, in the smart cities domain, quickly reacting within a few seconds to anomalous events in sensor streams of surveillance and security applications is of key importance.

Research contribution RCO2 – DIVIDE component enabling the automatic & adaptive configuration of the conditions & window parameters of queries based on use case context, and enabling privacy by design

The second contribution of this dissertation addresses research challenge RCH2. It integrates adaptiveness and privacy by design into the cascading reasoning framework that is the result from the first research contribution. It presents a semantic IoT platform component that can *adaptively* derive and configure the stream reasoning tasks on the edge components in an IoT network. This component is called DIVIDE. DIVIDE is adaptive and context-aware by design, as it automatically ensures that the platform's local stream processing components are always evaluating those queries that are contextually relevant according to the current use case context. Its design ensures that the queries can be evaluated in a performant way, also on devices with fewer resources that are often present in the edge of IoT networks. The component can be configured in such a way that queries can be deployed that have adaptive, context-aware query conditions and window parameters. Moreover, the DIVIDE component enables privacy by design: it allows end users to integrate privacy by design into applications by giving them the flexibility to define, on different levels of abstraction, which parts of the data can be sent over the network and which data is kept locally.

This research contribution will investigate the following hypotheses:

Research hypothesis RH3: The methodological design of a semantic IoT platform component that derives and configures the conditions & window parameters of stream processing queries whenever the use case context changes will result in adaptive, context-aware queries that only require simple filtering and thus enable the local filtering of contextually relevant events in less than 5 seconds on low-end IoT devices with few resources. This will fully remove the required manual query reconfiguration effort when changes to the use case context occur.

Research hypothesis RH4: The methodological design of a semantic IoT platform component that enables privacy by design will let the end user in 100% control about which data abstractions can be sent over the network and which data is not leaving the local environments of the IoT network, while maintaining an overhead to adapt the queries based on changing use case context that is at most 1 order of magnitude (i.e., 10 times) higher than the execution time of semantic queries on equivalent state-of-the-art real-time reasoning set-ups.

Considering research hypothesis RH3, the threshold of 5 seconds is relevant to different IoT application domains. For example, in homecare use cases in the healthcare domain, it is important that generated alarms or calls made by a patient in their home are handled by call operators in a timely fashion. To ensure that a call operator has access to the required real-time information to properly assess incoming alarms, such as real-time anomalies in the behavior of the patient, supportive real-time monitoring tasks, such as the monitoring of activity patterns, should be executed in a performant way, i.e., in at most 5 seconds. As another example, consider the smart cities domain with various cases requiring time efficiency like surveillance and security applications. In industrial applications, certain local anomalies observed in manufacturing processes should also be quickly reported to the more central components in the pipeline to allow for appropriate reaction to these anomalies.

Research contribution RCO3 – Extension of the DIVIDE component enabling automated adaptation and distribution of the cascading reasoning across the IoT network based on situational context

The third contribution of this dissertation addresses research challenge RCH3. It extends the semantic IoT platform component from research contribution RCO2 by enabling it to be adaptive to the full environment in which stream processing tasks (queries) are deployed. Relevant environmental context does not only include the use case context, but also situational context: characteristics of the network, resource usage on the stream processing devices, properties of the data streams, and the real-time performance of the stream processing components. The generic design of the DIVIDE component makes it possible to monitor these and additional environmental context parameters, and automatically adapt the configuration

of query window parameters *and/or* the distribution of the queries (location) in the IoT network. The design allows end users to dynamically configure for every use case which situational context parameters should influence the configuration and distribution of queries, and in what way.

This research contribution will investigate the following hypothesis:

Research hypothesis RH5: The methodological design of a semantic IoT platform component that monitors the situational context will result in an adaptive system that can update the window parameter configuration and distribution (i.e., location) to varying situational context, *precisely* according to use case specific rules and thresholds as defined by the end user, for a realistic local data stream of at least 150 observations per second.

Considering research hypothesis RH5, the quantification of 150 observations per second is a realistic lower bound of the high data stream velocity in IoT applications. For example, in healthcare, this number can be confirmed by a representative open dataset that was collected in a smart home [41], where the velocity of a local data stream for a single patient is indeed higher than 150 sensor observations per second.

Research contribution RC04 – A semantic platform enabling context-aware & performant IoT applications on streaming IoT data

The final research contribution of this dissertation addresses research challenge RCH4. It focuses on closing the feedback loop in IoT applications by integrating DIVIDE into a full semantic platform together with other building blocks built on Semantic Web technologies. This results in a distributed, cascading reasoning reference architecture that can optimize relevant IoT use cases by providing data-driven semantic services and constructing cross-organizational workflows. These workflows can be dynamically constructed using a semantic workflow engine. By coupling semantic services and semantic workflows to the outputs of the stream processing components managed by DIVIDE, resulting actions and workflows could update the use case context which in turn triggers DIVIDE to adaptively update the context-aware queries.

This research contribution will investigate the following hypothesis:

Research hypothesis RH6: A semantic IoT platform component that adaptively manages and configures queries according to varying environmental context, can be embedded in a semantic platform with other semantic components that define and construct data-driven semantic services and cross-organizational semantic workflows. Put together, the resulting cascading reasoning architecture can be leveraged to optimize relevant IoT use cases.

Considered use cases for the evaluations

Throughout the research, the generic architecture of the cascading reasoning framework has been applied to and evaluated on four use cases in the healthcare domain. This has been done in collaboration with several companies and actors from the healthcare domain, such as Televic Healthcare, a company specialized in nurse call systems, Z-Plus, a company facilitating homecare, Ghent University Hospital, and many more. In what follows, the four use cases considered for the evaluations in this dissertation are very briefly introduced. For each of those four use cases, Table 1.2 provides an overview of which research contributions are evaluated using this use case.

- **Use case UC1: Hospital monitoring**
  This use case focuses on the pervasive monitoring of hospitalized patients. It considers a hospital with medically diagnosed patients that are hospitalized in ambient-intelligent care rooms equipped with various sensors and actuators. The patients are constantly monitored based on their medical diagnoses to detect any alarming situations as they occur, and to responsively react to them by actuating on the environment and alerting a nurse if required.
- **Use case UC2: Homecare monitoring**
  This use case focuses on the continuous monitoring in homecare of elderly patients in different service flats spread out over the network. These service flats are equipped with a variety of sensors and an alarm system for the patients to generate an alarm to a care center whenever they are in need of assistance. Different monitoring tasks are relevant to this homecare environment, in order to continuously assess the condition of the patient. Multiple examples are addressed in this dissertation, including the monitoring of in-home activity patterns that are either part of the routine of the patient or not, to detect anomalies in the patient's daily life pattern, and the smart, personalized monitoring of relevant (medical) parameters to detect specific alarming situations occurring at the patient.
- **Use case UC3: Headache monitoring**
  This use case focuses on the continuous monitoring of headache patients. The use case is related to the mBrain study, which is specifically about the continuous follow-up of patients diagnosed with a primary headache disorder such as migraine or cluster headache. The use case mainly considers the monitoring of headache symptoms while the patient is experiencing a headache attack, and the monitoring of possible headache triggers. The monitoring is primarily based on monitored events from a smartphone application and predicted events from wearable and smartphone sensors.
- **Use case UC4: Cyclist monitoring**
  This use case focuses on the continuous monitoring of amateur cyclists while

they are riding their bike. More specifically, the use case considers the monitoring of the cyclist's heart rate to give personalized, real-time feedback to the cyclist on their heart rate and heart rate training zones, in order to have them perform the most efficient training.

Use cases UC1, UC2 and UC3 all have to deal with complex medical domain knowledge. This complexity is especially challenging in use case UC1 and some applications of use case UC2. In addition, while high-velocity data streams can be a real challenge in all use cases, this challenge is the biggest in use case UC2. Use case UC3 is different to the other use cases because of its mobile nature and because it also operates on data streams containing predicted or user-generated events instead of only raw sensor data. In use case UC4, the complexity of the medical domain knowledge is lower, but the restrictions on responsiveness are more challenging. This is also largely due to the limitations of the device and platform set-up in this use case.

All of these use cases are designed in collaboration with domain experts. More details about these different use cases are presented in the appropriate chapters and appendices of this dissertation, as indicated in Table 1.2.

1.7 Outline

This doctoral dissertation consists of seven main chapters, which include this introductory chapter and a concluding chapter, and two appendices. This section presents a brief overview of the contents of the individual chapters and appendices in relation to the research challenges and contributions of this dissertation. Table 1.1 summarizes the relation between the chapters and the research challenges & contributions. Table 1.2 gives an overview of which research contributions are evaluated by every use case considered for the evaluations in this dissertation, and in which chapter of the dissertation this evaluation is addressed. Figure 1.11 visually clarifies the positioning of the chapters & appendices of this dissertation, and illustrates how they are linked.

The chapters and appendices of this dissertation are composed of several publications that were realized within the scope of this PhD research. The selected publications provide an integral and consistent overview of the performed work. The complete list of peer-reviewed publications that resulted from this PhD research is presented in Section 1.8.

Chapter 2 presents a generic cascading reasoning framework across an IoT network that is built upon Semantic Web technologies. This architecture also fits within the reference architecture for AAL and ELE environments. The framework is applied to the pervasive healthcare use case UC1 about the responsive, ambient-intelligent monitoring of hospitalized patients. The chapter evaluates the overall and component-level performance of the resulting platform set-up, and discusses these

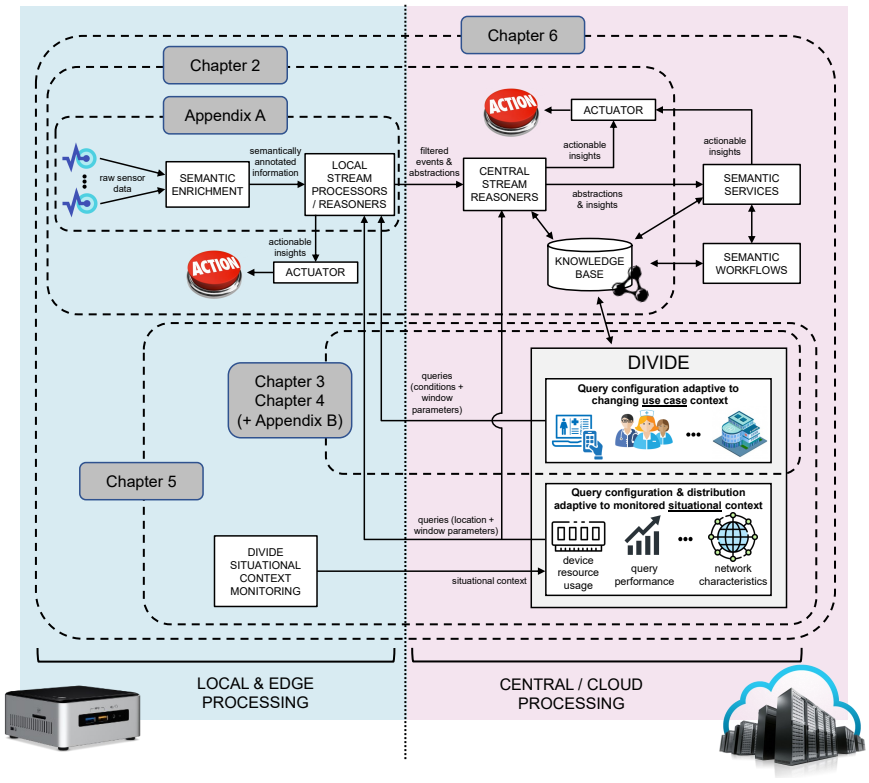


Figure 1.11: Schematic positioning of the different chapters and appendices in this dissertation, highlighting the coherence between the different contributions of this dissertation

Table 1.1: Overview of how every research challenge (RCH) and corresponding research contribution (RCO) is addressed in the different chapters of this dissertation

Research challenge Research contribution	RCH1 RCO1	RCH2 RCO2	RCH3 RCO3	RCH4 RCO4
Chapter 2	✘			
Chapter 3	✘	✘		
Chapter 4	✘	✘		
Chapter 5			✘	
Chapter 6				✘

Table 1.2: Overview of which research contributions are evaluated by every use case considered for the evaluations in this dissertation, and in which chapter of the dissertation this evaluation is addressed

Use case	Topic	Research contribution	Chapter
UC1	Hospital monitoring	RCO1	Chapter 2
UC2	Homecare monitoring	RCO1 & RCO2 RCO3 RCO4	Chapter 3 Chapter 5 Chapter 6
UC3	Headache monitoring	RCO1 & RCO2	Chapter 4 (+ Appendix B)
UC4	Cyclist monitoring	RCO1	Appendix A

results in relation to how the platform addresses other requirements in healthcare applications, with a focus on responsiveness and local autonomy. This chapter discusses research contribution RCO1 and addresses research challenge RCH1.

Appendix A zooms in on the local components of the cascading reasoning framework presented in Chapter 2 for the cyclist monitoring use case UC4. More specifically, it discusses a real-time feedback platform on low-end devices that performs personalized monitoring of heart rate and heart rate training zones in amateur cyclists.

Chapter 3 presents the semantic IoT platform component called DIVIDE. DIVIDE can adaptively derive and configure the context-aware queries for IoT data streams based on the actual use case context. By employing DIVIDE in a cascading reasoning architecture, DIVIDE enables privacy by design. The chapter presents the methodological design of DIVIDE, a Proof-of-Concept (PoC) implementation, and a performance evaluation of this implementation. This evaluation considers use case UC2 about the continuous homecare monitoring of elderly people, which specifically focuses in this chapter on monitoring the patients' in-home activity patterns. The chapter mainly discusses research contribution RCO2 by addressing research challenge RCH2. Moreover, it also still addresses research challenge RCH1 and research contribution RCO1 by integrating DIVIDE into the cascading reasoning architecture with a focus on performance.

Chapter 4 introduces the headache monitoring use case UC3 and discusses how the cascading reasoning framework with DIVIDE can be applied to it. The main purpose of this chapter is to illustrate the generic design of the designed solutions. Use case UC3 is related to the mBrain study, which considers the continuous follow-up of primary headache disorder patients. The chapter presents the design of the knowledge-driven services of the mBrain system, in which DIVIDE is employed to monitor contextual events during headache attacks and possible headache triggers, based on the use case context. **Appendix B** provides more in-depth details and context to the interested reader about the mBrain study and the headache monitoring use

case UC3 of Chapter 4. It discusses the details of the mBrain data collection set-up and zooms in on the design of the knowledge-driven, real-time classification system for individual headache attacks, which is also briefly touched upon in Chapter 4. It should be noted that this appendix does not contain any new research contributions, nor is it essential to understand the overall research presented in this dissertation.

Chapter 5 further extends the methodological design of the DIVIDE component, as introduced in Chapter 3, to continuously monitor situational context parameters. It introduces a distributed architecture of local and global monitoring subcomponents that allows end users to dynamically configure per use case how the situational context should influence the distribution of stream processing queries across the network and the configuration of their window parameters. To this end, the chapter also presents an ontology meta model to represent monitoring information and meta-information about the platform set-up. An implementation of the extended design of DIVIDE is discussed and evaluated on the homecare monitoring use case UC2 introduced in Chapter 3. The chapter discusses research contribution RCO3 by addressing research challenge RCH3.

Chapter 6 puts the research contributions presented in the previous chapters into broader context. It shows how DIVIDE can be embedded as a building block into a full semantic platform following a cascading reasoning architecture, *together* with other tools built upon Semantic Web technologies. It specifically zooms in on such a platform for the healthcare domain, considering the homecare monitoring use case UC2 to demonstrate how the platform can optimize continuous (home)care. It presents and evaluates a demonstrator for this use case, which mainly focuses in this chapter on the smart home monitoring of patients according to their medical diagnoses and the construction & cross-organizational coordination of semantic workflows representing patients' treatment plans. This way, the chapter discusses research contribution RCO4 by addressing research challenge RCH4.

To conclude this doctoral dissertation, **Chapter 7** summarizes the previous chapters and reflects on the different research challenges, contributions and hypotheses. In addition, remaining challenges and possible future research directions are identified.

1.8 Publications

The research results obtained during this PhD study have been published in scientific journals and presented at different international conferences. The section presents an overview of these publications.

1.8.1 Publications in international journals (A1, listed in the Science Citation Index[2])

1. **Mathias De Brouwer**, Femke Ongenae, Pieter Bonte, and Filip De Turck. *Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions*. Published in Sensors, Volume 18, Issue 10, October 2018. doi:10.3390/s18103514.

2. Esteban Municio, Glenn Daneels, **Mathias De Brouwer**, Femke Ongenae, Filip De Turck, Bart Braem, Jeroen Famaey, and Steven Latré. *Continuous Athlete Monitoring in Challenging Cycling Environments Using IoT Technologies*. Published in IEEE Internet of Things Journal, Volume 6, Issue 6, September 2019. doi:10.1109/JIOT.2019.2942761.

3. **Mathias De Brouwer**[3], Nicolas Vandenbussche[3], Bram Steenwinckel, Marija Stojchevska, Jonas Van Der Donckt, Vic Degraeve, Jasper Vaneessen, Filip De Turck, Bruno Volckaert, Paul Boon, Koen Paemeleire, Sofie Van Hoecke, and Femke Ongenae. *mBrain: Towards the Continuous Follow-up and Headache Classification of Primary Headache Disorder Patients*. Published in BMC Medical Informatics and Decision Making, Volume 22, March 2022. doi:10.1186/s12911-022-01813-w.

4. Marija Stojchevska, Bram Steenwinckel, Jonas Van Der Donckt, **Mathias De Brouwer**, Annelies Goris, Filip De Turck, Sofie Van Hoecke, and Femke Ongenae. *Assessing the Added Value of Context During Stress Detection From Wearable Data*. Published in BMC Medical Informatics and Decision Making, Volume 22, October 2022. doi:10.1186/s12911-022-02010-5.

5. Bram Steenwinckel, **Mathias De Brouwer**, Marija Stojchevska, Filip De Turck, Sofie Van Hoecke, and Femke Ongenae. *TALK: Tracking Activities by Linking Knowledge*. Published in Engineering Applications of Artificial Intelligence, Volume 122, March 2023. doi:10.1016/j.engappai.2023.106076.

6. **Mathias De Brouwer**, Bram Steenwinckel, Ziye Fang, Marija Stojchevska, Pieter Bonte, Filip De Turck, Sofie Van Hoecke, and Femke Ongenae. *Context-Aware Query Derivation for IoT Data Streams with DIVIDE Enabling Privacy By Design*. Published in Semantic Web, Volume 14, Issue 5, May 2023. doi:10.3233/SW-223281.

[2]The publications listed are recognized as "A1 publications", according to the following definition used by Ghent University: A1 publications are articles listed in the Science Citation Index Expanded, the Social Science Citation Index or the Arts and Humanities Citation Index of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper.

[3]The first two authors of this work contributed equally.

7. Marija Stojchevska, **Mathias De Brouwer**, Martijn Courteaux, Femke Ongenae, and Sofie Van Hoecke. *From Lab to Real World: Assessing the Effectiveness of Human Activity Recognition and Optimization through Personalization*. Published in Sensors, Volume 23, Issue 10, May 2023. doi:10.3390/s23104606.

8. **Mathias De Brouwer**, Filip De Turck, and Femke Ongenae. *Enabling Efficient Semantic Stream Processing across the IoT Network through Adaptive Distribution with DIVIDE*. Submitted for review to Journal of Network and Systems Management, June 2023.

9. **Mathias De Brouwer**, Pieter Bonte, Dörthe Arndt, Miel Vander Sande, Anastasia Dimou, Ruben Verborgh, Filip De Turck, and Femke Ongenae. *Optimized Continuous Homecare Provisioning through Distributed Data-Driven Semantic Services and Cross-Organizational Workflows*. Submitted for review to Journal of Biomedical Semantics, August 2023.

1.8.2 Publications in international conferences (P1, listed in the Science Citation Index[4])

1. **Mathias De Brouwer**, Femke Ongenae, Glenn Daneels, Esteban Municio, Jeroen Famaey, Steven Latré, and Filip De Turck. *Personalized Real-Time Monitoring of Amateur Cyclists on Low-End Devices: Proof-of-Concept & Performance Evaluation*. Published in the Companion Proceedings of The World Wide Web Conference 2018 (WWW 2018), Lyon, France, April 2018. doi:10.1145/3184558.3191648.

2. **Mathias De Brouwer**, Femke Ongenae, and Filip De Turck. *Demonstration of a Stream Reasoning Platform on Low-End Devices to Enable Personalized Real-Time Cycling Feedback*. Published in the The Semantic Web: ESWC 2019 Satellite Events, Portorož, Slovenia, June 2019. doi:10.1007/978-3-030-32327-1_6.

3. **Mathias De Brouwer**, Pieter Bonte, Dörthe Arndt, Miel Vander Sande, Pieter Heyvaert, Anastasia Dimou, Ruben Verborgh, Filip De Turck, and Femke Ongenae. *Distributed Continuous Home Care Provisioning through Personalized Monitoring & Treatment Planning*. Published in the Companion Proceedings of the Web Conference 2020 (WWW 2020), online, April 2020. doi:10.1145/3366424.3383528.

[4]The publications listed are recognized as "P1 publications", according to the following definition used by Ghent University: P1 publications are proceedings listed in the Conference Proceedings Citation Index – Science or Conference Proceedings Citation Index – Social Science and Humanities of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper, except for publications that are classified as A1.

1.8.3 Publications in other peer-reviewed conferences (C1)

1. **Mathias De Brouwer**, Dörthe Arndt, Pieter Bonte, Filip De Turck, and Femke Ongenae. *DIVIDE: Adaptive Context-Aware Query Derivation for IoT Data Streams*. Published in the Joint Proceedings of the International Workshops on Sensors and Actuators on the Web, and Semantic Statistics, co-located with the 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 2019. Available from: https://ceur-ws.org/Vol-2549/article-01.pdf.

2. Pieter Bonte, **Mathias De Brouwer**, Dörthe Arndt, Miel Vander Sande, Pieter Heyvaert, Anastasia Dimou, Pieter Colpaert, Ruben Verborgh, Filip De Turck, and Femke Ongenae. *Context-Aware Route Planning: A Personalized and Situation-Aware Multi-Modal Transport Routing Approach*. Published in the Proceedings of the ISWC 2020 Demos and Industry Tracks: From Novel Ideas to Industrial Practice, co-located with the 19th International Semantic Web Conference (ISWC 2020), online, November 2020. Available from: http://ceur-ws.org/Vol-2721/paper499.pdf.

3. **Mathias De Brouwer**, Nicolas Vandenbussche, Bram Steenwinckel, Marija Stojchevska, Jonas Van Der Donckt, Vic Degraeve, Filip De Turck, Koen Paemeleire, Sofie Van Hoecke, and Femke Ongenae. *Towards Knowledge-Driven Symptom Monitoring & Trigger Detection of Primary Headache Disorders*. Published in the Companion Proceedings of the Web Conference 2022 (WWW 2022), online, April 2022. doi:10.1145/3487553.3524256.

4. Jonas Van Der Donckt[5], **Mathias De Brouwer**[5], Pieter Moens, Marija Stojchevska, Bram Steenwinckel, Stef Pletinck, Nicolas Vandenbussche, Annelies Goris, Koen Paemeleire, Femke Ongenae, and Sofie Van Hoecke. *From Self-Reporting to Monitoring for Improved Migraine Management*. Published in the Proceedings of the 1st RADar Conference on Engineer Meets Physician (EmP 2022), Roeselare, Belgium, May 2022. Available from: http://hdl.handle.net/1854/LU-01GT2DJEQPMTP44D8XNQHAF6GG.

5. Bram Steenwinckel, **Mathias De Brouwer**, Marija Stojchevska, Jeroen Van Der Donckt, Jelle Nelis, Joeri Ruyssinck, Joachim van der Herten, Koen Casier, Jan Van Ooteghem, Pieter Crombez, Filip De Turck, Sofie Van Hoecke, and Femke Ongenae. *Data Analytics For Health and Connected Care: Ontology, Knowledge Graph and Applications*. Published in the Proceedings of the 16th EAI International Conference on Pervasive Computing Technologies for Healthcare (EAI PervasiveHealth 2022), Thessaloniki, Greece, December 2022. doi:10.1007/978-3-031-34586-9_23.

[5]The first two authors of this work contributed equally.

References

[1] K. Rose, S. Eldridge, and L. Chapin. *The Internet of Things: An overview*. Technical report, The Internet Society (ISOC), 2015. Available from: https://www.internetsociety.org/wp-content/uploads/2017/08/ISOC-IoT-Overview-20151221-en.pdf.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. *Internet of Things: A survey on enabling technologies, protocols, and applications*. IEEE Communications Surveys & Tutorials, 17(4):2347–2376, 2015. doi:10.1109/COMST.2015.2444095.

[3] Y. Perwej, K. Haq, F. Parwej, M. Mumdouh, and M. Hassan. *The Internet of Things (IoT) and its application domains*. International Journal of Computer Applications, 182(49), 2019. doi:10.5120/IJCA2019918763.

[4] Transforma Insights. *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030*, 2022. Accessed: 2023-04-30. Available from: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[5] Y. A. Qadri, A. Nauman, Y. B. Zikria, A. V. Vasilakos, and S. W. Kim. *The Future of Healthcare Internet of Things: A Survey of Emerging Technologies*. IEEE Communications Surveys & Tutorials, 22(2):1121–1167, 2020. doi:10.1109/COMST.2020.2973314.

[6] Precedence Research. *Internet of Things (IoT) in Healthcare Market Report*, 2022. Accessed: 2023-04-30. Available from: https://www.precedenceresearch.com/internet-of-things-in-healthcare-market.

[7] K. Singh, K. Kaushik, Ahatsham, and V. Shahare. *Role and Impact of Wearables in IoT Healthcare*. In Proceedings of the Third International Conference on Computational Intelligence and Informatics, pages 735–742. Springer Singapore, 2020. doi:10.1007/978-981-15-1480-7_67.

[8] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. *Semantics for the Internet of Things: Early Progress and Back to the Future*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012. doi:10.4018/jswis.2012010101.

[9] F. Ongenae, P. Duysburgh, N. Sulmon, M. Verstraete, L. Bleumers, S. De Zutter, S. Verstichel, A. Ackaert, A. Jacobs, and F. De Turck. *An ontology co-design method for the co-creation of a continuous care ontology*. Applied Ontology, 9(1):27–64, 2014. doi:10.3233/AO-140131.

[10] G. Marques, A. K. Bhoi, and K. Hareesha, editors. *IoT in Healthcare and Ambient Assisted Living*. Springer Singapore, 2021. doi:10.1007/978-981-15-9897-5.

[11] R. Maskeliūnas, R. Damaševičius, and S. Segal. *A review of Internet of Things technologies for Ambient Assisted Living environments*. Future Internet, 11(12), 2019. doi:10.3390/fi11120259.

[12] A. Dohr, R. Modre-Opsrian, M. Drobics, D. Hayn, and G. Schreier. *The Internet of Things for Ambient Assisted Living*. In 2010 Seventh International Conference on Information Technology: New Generations, pages 804–809, 2010. doi:10.1109/ITNG.2010.104.

[13] R. I. Goleva, N. M. Garcia, C. X. Mavromoustakis, C. Dobre, G. Mastorakis, R. Stainov, I. Chorbev, and V. Trajkovik. *Chapter 8 – AAL and ELE Platform Architecture*. In C. Dobre, C. Mavromoustakis, N. Garcia, R. Goleva, and G. Mastorakis, editors, Ambient Assisted Living and Enhanced Living Environments: Principles, Technologies and Control, pages 171–209. Butterworth-Heinemann, 2017. doi:10.1016/B978-0-12-805195-5.00008-9.

[14] N. M. Garcia and J. J. P. C. Rodrigues, editors. *Ambient Assisted Living*. CRC Press, 2015. doi:10.1201/b18520.

[15] R. I. Goleva, I. Ganchev, C. Dobre, N. Garcia, and C. Valderrama, editors. *Enhanced Living Environments: From models to technologies*. Institution of Engineering and Technology, 2017. doi:10.1049/PBHE010E.

[16] D. Corral-Plaza, I. Medina-Bulo, G. Ortiz, and J. Boubeta-Puig. *A stream processing architecture for heterogeneous data sources in the Internet of Things*. Computer Standards & Interfaces, 70, 2020. doi:10.1016/j.csi.2020.103426.

[17] T. Atanasova. *Methods for Processing of Heterogeneous Data in IoT Based Systems*. In DCCN 2019: Distributed Computer and Communication Networks, pages 524–535, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-36625-4_42.

[18] S. Zeadally, F. Siddiqui, Z. Baig, and A. Ibrahim. *Smart healthcare: Challenges and potential solutions using internet of things (IoT) and big data analytics*. PSU Research Review, 4(2):149–168, 2020. doi:10.1108/PRR-08-2019-0027.

[19] V. Jagadeeswari, V. Subramaniyaswamy, R. Logesh, and V. Vijayakumar. *A study on medical Internet of Things and Big Data in personalized healthcare system*. Health Information Science and Systems, 6, 2018. doi:10.1007/s13755-018-0049-x.

[20] V.-D. Ta, C.-M. Liu, and G. W. Nkabinde. *Big data stream computing in healthcare real-time analytics*. In 2016 IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), pages 37–42, 2016. doi:10.1109/ICCCBDA.2016.7529531.

[21] X. Su, E. Gilman, P. Wetz, J. Riekki, Y. Zuo, and T. Leppänen. *Stream reasoning for the Internet of Things: Challenges and gap analysis*. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), pages 1–10, New York, NY, USA, 2016. Association for Computing Machinery (ACM). doi:10.1145/2912845.2912853.

[22] J. Rowley. *The wisdom hierarchy: representations of the DIKW hierarchy*. Journal of Information Science, 33(2):163–180, 2007. doi:10.1177/0165551506070706.

[23] X. Su, J. Riekki, J. K. Nurminen, J. Nieminen, and M. Koskimies. *Adding semantics to Internet of Things*. Concurrency and Computation: Practice and Experience, 27(8):1844–1860, 2015. doi:10.1002/cpe.3203.

[24] T. R. Gruber. *A translation approach to portable ontology specifications*. Knowledge Acquisition, 5(2):199–220, 1993. doi:10.1006/knac.1993.1008.

[25] W3C OWL Working Group. *OWL 2 Web Ontology Language*. W3C Recommendation, World Wide Web Consortium (W3C), 2012. Available from: https://www.w3.org/TR/owl2-overview/.

[26] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride. *RDF 1.1 concepts and abstract syntax*. W3C Recommendation, World Wide Web Consortium (W3C), 2014. Available from: https://www.w3.org/TR/rdf11-concepts/.

[27] S. Harris and A. Seaborne. *SPARQL 1.1 Query Language*. W3C Recommendation, World Wide Web Consortium (W3C), 2013. Available from: https://www.w3.org/TR/sparql11-query/.

[28] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. Van Harmelen. *Towards expressive stream reasoning*. In Semantic Challenges in Sensor Networks, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. doi:10.4230/DagSemProc.10042.4.

[29] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. *OWL 2 Web Ontology Language Profiles (Second Edition)*. W3C Recommendation, World Wide Web Consortium (W3C), 2012. Available from: https://www.w3.org/TR/owl2-profiles/.

[30] P. Bonte, F. Ongenae, and F. De Turck. *Subset reasoning for event-based systems*. IEEE Access, 7:107533–107549, 2019. doi:10.1109/ACCESS.2019.2932937.

[31] K. Donnelly. *SNOMED-CT: The advanced terminology and coding system for eHealth*. Studies in Health Technology and Informatics, 121, 2006.

[32] SNOMED International. *SNOMED*. Accessed: 2023-04-14. Available from: https://www.snomed.org.

[33] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. *Stream reasoning: A survey and outlook*. Data Science, 1(1-2):59–83, 2017. doi:10.3233/DS-170006.

[34] G. Cugola and A. Margara. *Processing Flows of Information: From Data Stream to Complex Event Processing*. ACM Computing Surveys, 44(3), 2012. doi:10.1145/2187671.2187677.

[35] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. *C-SPARQL: a continuous query language for RDF data streams*. International Journal of Semantic Computing, 4(1):3–25, 2010. doi:10.1142/S1793351X10000936.

[36] J.-P. Calbimonte, O. Corcho, and A. J. Gray. *Enabling ontology-based access to streaming data sources*. In The Semantic Web – ISWC 2010, pages 96–111. Springer, 2010. doi:10.1007/978-3-642-17746-0_7.

[37] M. A. Sahi, H. Abbas, K. Saleem, X. Yang, A. Derhab, M. A. Orgun, W. Iqbal, I. Rashid, and A. Yaseen. *Privacy Preservation in e-Healthcare Environments: State of the Art and Future Directions*. IEEE Access, 6:464–478, 2018. doi:10.1109/ACCESS.2017.2767561.

[38] A. Cavoukian. *Privacy by design*, 2009. Accessed: 2022-09-25. Available from: https://www.ipc.on.ca/wp-content/uploads/Resources/7foundationalprinciples.pdf.

[39] K. Cao, Y. Liu, G. Meng, and Q. Sun. *An Overview on Edge Computing Research*. IEEE Access, 8:85714–85728, 2020. doi:10.1109/ACCESS.2020.2991734.

[40] S. H. and N. V. *A Review on Fog Computing: Architecture, Fog with IoT, Algorithms and Research Challenges*. ICT Express, 7(2):162–176, 2021. doi:10.1016/j.icte.2021.05.004.

[41] B. Steenwinckel, M. De Brouwer, M. Stojchevska, J. Van Der Donckt, J. Nelis, J. Ruyssinck, J. van der Herten, K. Casier, J. Van Ooteghem, P. Crombez, F. De Turck, S. Van Hoecke, and F. Ongenae. *DAHCC: Data Analytics For Health and Connected Care: Connecting Data Analytics to Healthcare Knowledge in an IoT environment*, 2022. Accessed: 2022-02-03. Available from: https://dahcc.idlab.ugent.be.

2

Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions

The vision of cascading reasoning allows moving away from the centralized processing architectures that typically exist in healthcare or other IoT applications. It can be aligned with the vision of edge computing by involving heterogeneous processing devices in the local and edge parts of IoT networks. This way, cascading reasoning has the potential to solve the issues with centralized solutions, which have performance issues and lack local autonomy. However, the vision of cascading reasoning has not yet been fully realized in a generic semantic framework that is applicable to healthcare and other IoT application domains. Therefore, this chapter solves this by presenting such a cascading reasoning framework. The hospital monitoring use case UC1 is chosen to apply the framework and evaluate its performance. In Appendix A, the local components of the framework are employed for use case UC4 about performing personalized real-time monitoring of amateur cyclists.

This chapter addresses research challenge RCH1 ("Performant & responsive real-time stream reasoning with local autonomy across a heterogeneous IoT network") by discussing research contribution RC01. It validates research hypothesis RH1: "The realization of a generic cascading reasoning framework in an IoT network will improve the overall performance of semantic stream reasoning on IoT data streams. The full pipeline of stream reasoning components will be able to generate relevant actionable insights from events in the data and handle those events in less than 5 seconds.". Moreover, this chapter also validates research hypothesis RH2: "The realization of a generic

cascading reasoning framework in an IoT network will introduce local autonomy by letting local & edge devices in the network host queries. This will allow certain events in the data to be handled locally through actionable insights derived from the data, without requiring human intervention or involving central reasoning components. The local & edge components in the pipeline will also be able to perform these tasks in less than 5 seconds.".

⋆ ⋆ ⋆

M. De Brouwer, F. Ongenae, P. Bonte, and F. De Turck

Published in Sensors Journal, Volume 18, Issue 10, October 2018.

Abstract
In hospitals and smart nursing homes, ambient-intelligent care rooms are equipped with many sensors. They can monitor environmental and body parameters, and detect wearable devices of patients and nurses. Hence, they continuously produce data streams. This offers the opportunity to collect, integrate and interpret this data in a context-aware manner, with a focus on reactivity and autonomy. However, doing this in real time on huge data streams is a challenging task. In this context, cascading reasoning is an emerging research approach that exploits the trade-off between reasoning complexity and data velocity by constructing a processing hierarchy of reasoners. Therefore, a cascading reasoning framework is proposed in this chapter. A generic architecture is presented allowing to create a pipeline of reasoning components hosted locally, in the edge of the network, and in the cloud. The architecture is implemented on a pervasive health use case, where medically diagnosed patients are constantly monitored, and alarming situations can be detected and reacted upon in a context-aware manner. A performance evaluation shows that the total system latency is mostly lower than 5 s, allowing for responsive intervention by a nurse in alarming situations. Using the evaluation results, the benefits of cascading reasoning for healthcare are analyzed.

2.1 Background

2.1.1 Introduction

The ultimate ambient-intelligent care rooms of the future in smart hospitals or smart nursing homes consist of a wide range of Internet of Things (IoT) enabled devices equipped with a plethora of sensors, which constantly generate data [1, 2]. Wireless Sensor Networks (WSNs) can be used to monitor environmental parameters, such as light intensity and sound, and Body Area Networks (BANs) can monitor vital body parameters, such as heart rate, blood pressure or body temperature. Other IoT-enabled devices allow for performing indoor positioning, to detect when doors or windows are opened, or to discover if a patient is lying in bed or sitting in a couch. Intelligent smart

home IoT devices can be used to take control of and automate the lighting, window blindings, heating, ventilation and air conditioning (HVAC), and more. Moreover, domain and background knowledge contains information on diseases, medical symptoms, the patients' Electronic Health Record (EHR), and much more. The advantage of the IoT is that the data streams originating from the various sensors and devices can be combined with this knowledge to derive new knowledge about the environment and the patient's current condition [3]. This enables devices to achieve situation- and context-awareness, and enables better support of the nursing staff in their activities [4].

Consider the example of a pervasive health context in which a patient suffers from a concussion. Medical domain knowledge states that concussion patients are sensitive to light and sound. This knowledge can be combined with data streams coming from the light and sound sensors in the patient's room, to derive when an alarming situation occurs, i.e., when the patient is in his room and certain light or sound thresholds are crossed. When such an alarming situation is detected, automatic action can be taken, such as autonomously dimming the lights or alerting a caregiver. This can help to increase the comfort of both the patients and nurses, and help nurses to operate more efficiently.

By 2020, 20 to 30 billion IoT devices are forecasted to be in use worldwide within healthcare [5]. The data streams generated by these IoT devices are not only voluminous, but are also a heterogeneous, possibly noisy and incomplete set of time-varying data events [6]. As such, it is a challenging task to integrate, interpret and analyze the data streams on the fly to derive actionable insights.

Semantically enriching the data facilitates the consolidation of these data streams [7]. It imposes a common, machine-interpretable data representation. It also makes the properties of the device and the context in which the data was gathered explicit [7]. Moreover, it enables the integration of these streams with the domain and background knowledge.

Semantic Web technologies, such as the Resource Description Framework (RDF) and the Web Ontology Language (OWL), allow for achieving this semantic enrichment by using ontologies [7]. An ontology is a semantic model that formally describes the concepts in a particular domain, their relationships and attributes [8]. Using an ontology, heterogeneous data can be modeled in a uniform way. Recently, ontologies for the IoT have emerged, such as the Semantic Sensor Network (SSN) ontology [9], which facilitate the enrichment of IoT data. Moreover, prevalent healthcare ontologies exist, such as SNOMED [10] and FHIR [11], which model a lot of medical domain knowledge. By using the Linked Data approach [12], the semantic IoT data can then easily be linked to such domain knowledge and resources described by these and other models. Semantic reasoners, e.g., Hermit [13] and RDFox [14], have been designed to interpret this semantic interconnected data in order to derive useful knowledge [15], i.e., additional new implicit knowledge that can be useful for applications. For example, in the case of the concussion patient, a semantic reasoner

can automatically derive that the patient is sensitive to light and sound, and that light and sound sensors in the patient's room should be monitored. Based on the definitions of alarming situations in the ontology, the reasoner can infer from the data streams when such a situation occurs.

The complexity of semantic reasoning depends on the expressivity of the underlying semantic language, i.e., the expressivity of the ontology [16]. Different sublanguages exist, ranging from RDFS, which supports only simple statements, e.g., class inheritance, to OWL 2 DL, which supports expressive reasoning, e.g., cardinality restrictions on properties of classes. In OWL, it is assumed that any instance of a class may have an arbitrary number (zero or more) of values for a particular property. According to the W3C definition, cardinality constraints can be used to require a minimum number of values for that property, to allow only a specific number of values for that property, or to specify an exact number of values for that property. For example, in healthcare, it could be defined that an `Observation` is made by exactly 1 `Sensor`. Other profiles, such as OWL 2 RL, OWL 2 QL and OWL 2 EL, are situated in between RDFS and OWL 2 DL, trading off reasoning expressivity for computational efficiency. More expressive reasoning allows for deriving more interesting information from the streams and transforms it to actionable insights. In healthcare, high expressivity of the reasoner is required. In the example of the concussion patient, alarming situations can have complex definitions, making it impossible for less expressive reasoners to infer their occurrence. For example, an RDFS reasoner would not be able to infer that a patient with a concussion is sensitive to light and sound.

Semantic reasoning over large or complex ontologies is computationally intensive and slow. Hence, it cannot keep up with the velocity of large data streams generated in healthcare to derive real-time knowledge [15]. However, in healthcare, making decisions often is time-critical. For example, alarming situations for a patient should be reacted upon in a responsive manner. In this case, real-time means within a time frame of 5 s. For each situation or use case, real-time can be defined differently. In addition, available resources are limited, making the computational complexity of expressive reasoning for healthcare even a bigger issue. Moreover, when constructing a solution for these problems, privacy management of the patient data is an important consideration [17].

To tackle the issue with performing real-time analysis, two research trends have emerged, being stream reasoning and cascading reasoning. Stream reasoning [15] tries to incorporate semantic reasoning techniques in stream processing techniques. It defines a data stream as a sequence of time-annotated items ordered according to temporal criteria, and studies the application of inference techniques to such streaming data [15]. Cascading reasoning [18] exploits the trade-off between reasoning complexity and data stream velocity by constructing a processing hierarchy of reasoners. Hence, there is the need for a platform using these techniques to solve the issues in smart healthcare.

2.1.2 Objective and chapter organization

The objective of this chapter is the realization of a generic cascading reasoning platform, and the study of its applicability to solve the aforementioned smart healthcare issues. The cascading reasoning platform is implemented in an open flexible way, to make it easily extensible and applicable to different use cases. A Proof-of-Concept (PoC) application is implemented on a use case situated in pervasive healthcare. Using the PoC implementation, the performance of the framework is evaluated, and its advantages and disadvantages are discussed.

The remainder of this chapter is organized as follows. Section 2.2 discusses the related work. In Section 2.3, the general architecture of the proposed cascading reasoning platform is described. Sections 2.4 and 2.5 address the use case for the PoC and its implementation using the architecture components. Section 2.6 then describes the evaluation set-up, including the different evaluation scenarios and hardware set-up. Results of this evaluation are presented in Section 2.7. In Section 2.8, the evaluation results, advantages and disadvantages of the platform for the PoC use case are further discussed. Finally, Section 2.9 concludes the main findings and highlights future work.

2.2 Related work

2.2.1 Stream reasoning

Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) systems allow to query homogeneous streaming data structured according to a fixed data model [19]. However, in contrast to Semantic Web reasoners, DSMS and CEP systems are not able to deal with heterogeneous data sources and lack support for the integration of domain knowledge in a standardized fashion.

Therefore, stream reasoning [15] has emerged as a challenging research area that focuses on the adoption of semantic reasoning techniques for streaming data. The first prototypes of RDF Stream Processing (RSP) engines [20] mainly focus on stream processing. The most well-known examples of RSP engines are C-SPARQL [21] and CQELS [22], but others also exist, such as EP-SPARQL [23] and SPARQL-Stream [24]. Because a continuous data stream has no defined ending, a window is placed on top of the data stream. A continuous query is registered once and produces results continuously over time as the streaming data passes through the window. As such, these RSP engines can filter and query a continuous flow of data and can provide real-time answers. Each of these engines has different semantics and is tailored towards different use cases. Other solutions, e.g., Sparkwave [25] and INSTANS [26], use extensions of the RETE algorithm [27] for pattern matching.

As shown in Table 2.1, all considered RSP engines, except INSTANS, support integration of domain knowledge in the querying process. However, their reasoning

Table 2.1: Reasoning support in state-of-the-art RDF Stream Processing (RSP) engines

	Background Knowledge	Reasoning Capabilities
C-SPARQL	Yes	RDFS
SPARQLStream	Yes	None
EP-SPARQL	Yes	RDFS (in Prolog)
CQELS	Yes	None
Sparkwave	Yes	RDFS subset
INSTANS	No	None

capabilities are limited. None of the proposed systems is able to efficiently perform expressive OWL 2 DL reasoning on streaming data, which is often required for complex application domains such as healthcare.

StreamRule [28] is a 2-tier approach, combining stream processing with rule-based non-monotonic Answer Set Programming (ASP) to enable reasoning over data streams. However, ASP is not standardized, meaning no existing healthcare vocabularies can be exploited, in contrast to OWL.

In summary, stream reasoning tries to adopt semantic reasoning techniques for streaming data, but still lacks the possibility to support real-time and expressive reasoning at the same time. The existing available RSP engines aim at filtering and querying of streaming data, but they lack support for complex reasoning. To perform such complex reasoning, traditional semantic reasoners need to be used. However, this complex reasoning is computationally intensive and not capable of handling streaming data.

2.2.2 Cascading reasoning

The concept of cascading reasoning [29] was proposed to exploit the trade-off between reasoning complexity and data stream velocity. The aim is to construct a processing hierarchy of reasoners. At lower levels, high frequency data streams are filtered with little or no reasoning, to reduce the volume and rate of the data. At higher levels, more complex reasoning is possible, as the change frequency has been further reduced. This approach avoids feeding high frequency streaming data directly to complex reasoning algorithms. In the vision of cascading reasoning, streams are first fed to one or more RSP engines, and then to more expressive semantic reasoners.

The concept of cascading reasoners fits nicely with the current trend in IoT architectures towards Fog computing [30], where the edge is introduced as an intermediate layer between data acquisition and the cloud-based processing layer. The edge allows filtering and aggregation of data, resulting in reduced network congestions, less latency and improved scalability. In addition, it enables to process the data close to its source, which in turn can improve the response time, as it allows to rapidly act on events occurring in the environment. As such, fast and possibly less accurate derivations and

actions can be made at the edge of the network. These intermediate results can then be combined and forwarded to the cloud for further, more complex and less time-stringent processing. In this way, more privacy is also enabled. For example, these intermediate results can filter out sensitive data, to avoid sending it over the network.

Recently, much research effort has been put into Fog computing [31, 32], and Fog computing frameworks, such as FogFrame [33], are being designed. These frameworks focus on the creation of a dynamic software environment to execute services in a distributed way. They are useful for system deployment, execution and monitoring, but not sufficient to support cascading reasoning. In addition, a generic framework is required that enables cascading reasoning across the IoT fog.

Different distributed semantic reasoning frameworks exist, e.g., DRAGO, LarKC, Marvin and WebPIE [34]. However, they all have limitations in terms of applicability for the IoT [34]. First, they do not consider the heterogeneous nature of an IoT network. In particular, the use of low-end devices and networking aspects are not considered. Both criteria are, however, essential to Fog computing. Second, as they distribute reasoning evenly across nodes, cascading reasoning is not considered. Third, they do not focus on complex reasoning. Hence, these frameworks cannot be used as such.

Recent stream reasoning research also touches the area of Fog computing and devices with limited resources. Various relevant topics are addressed, from publishing RDF streams from smartphones [35], over optimizing semantic reasoning in memory-constrained environments [36], to optimizing the format to exchange and query RDF data in these environments [37]. These results can be useful for a cascading reasoning system, but do not solve the need for a generic cascading reasoning framework.

2.2.3 Frameworks for healthcare and Ambient Assisted Living

Within healthcare, Ambient Assisted Living (AAL) solutions offer IT products, services and systems that focus on the improvement of an individual's Quality of Life [38]. Enhanced Living Environments (ELE) include all technological achievements that support AAL environments [39].

Goleva et al. have presented a generic reference architecture for AAL and ELE platforms [40]. This architecture defines an AAL/ELE system as a distributed heterogeneous network. It supports the envisioned AAL as a Service (AALaaS) and ELE as a Service (ELEaaS) [39]. The architecture incorporates the Fog computing principles, by making a distinction between services running in the edge and in the cloud. Locally, sensor networks collect all data, which is transferred through the network. At Fog computing level, the architecture consists of regional components that perform local computation and storage. The goal of this generic reference architecture is to allow data processing to be done at different levels, depending on security, privacy, end-user preferences, technology, legislation, energy requirements etc. It also enables

real-time processing for critical data. As such, this generic architecture perfectly fits with the idea of deploying a cascading reasoning system across the IoT fog.

Table 2.2 compares the most prevalent state-of-the-art AAL frameworks and solutions [41–51] to the solution presented in this work. Many of these approaches are still cloud-based. However, approaches using Fog computing principles, where AAL system components are distributed across heterogeneous devices, are being incorporated more and more. Most existing AAL solutions specifically focus on the incorporation of semantic components in AAL systems. They all make use of reasoning techniques in some way, but stream reasoning is not tackled by any of them. Cascading reasoning is only tackled by one approach, ERMHAN [50], which distributes knowledge and reasoning across two nodes. In summary, there currently does not exist a semantic AAL solution that uses stream reasoning techniques in a cascading fashion across the IoT fog.

2.2.4 Contribution

The contribution of this work is the realization of the vision of cascading reasoning through a framework, using stream reasoning techniques and following the Fog computing principles. Stream reasoning techniques are required in order to infer real-time knowledge and actions from the voluminous and heterogeneous background knowledge and streaming data sources. However, current stream reasoning solutions fail to combine real-time and expressive reasoning. Incorporating them in a cascading fashion is a possible solution. In addition, cascading reasoning and Fog computing principles offer the potential to solve the existing issues in smart healthcare. These concepts have not yet been combined for AAL in previous works, as is shown in Table 2.2. Therefore, the contribution of this work is to combine them in a framework.

In concrete, the contribution of this work is threefold. First, an architecture is designed for this framework that fits within the generic reference architecture for AAL and ELE platforms [40]. Second, a concrete PoC implementation of this architecture for a specific pervasive healthcare use case is performed. Third, an evaluation of this PoC is conducted using three simulation scenarios.

2.3 Architecture of the cascading reasoning framework

The architecture of the generic cascading reasoning framework consists of four main components: an Observation Unit (OBU), an RSP Service (RSPS), a Local Reasoning Service (LRS) and a Back-end Reasoning Service (BRS). Furthermore, a central knowledge base is available. An overview of this architecture is given in Figure 2.1.

The central knowledge base contains the domain ontologies and static context information. For example, in healthcare, the knowledge base includes information on existing medical domain knowledge and a semantic version of the EHR of patients.

Table 2.2: Overview of the properties of the most prevalent state-of-the-art Ambient Assisted Living (AAL) frameworks and solutions, compared to the solution presented in Chapter 2

	Semantics	Reasoning Expressivity	Cascading Reasoning	Stream Reasoning	Fog Computing Principles
This work (Chapter 2)	✔	OWL DL & OWL RL	✔	✔	✔
R-Core [41]		rule-based contextual reasoning			✔
Valencia et al. [42]		case-based reasoning			
Nguyen et al. [43]		multi-objective reasoning			
IoT-SIM [44]	✔	ontology-based *(unspecified)*			✔
SeDan [45]	✔	plausible reasoning			
OCarePlatform [46]	✔	OWL DL			
Lassiera et al. [47]	✔	ontology-based *(unspecified)*			✔
Kuijs et al. [48]	✔	OWL DL			
CoCaMAAL [49]	✔	ontology-based *(unspecified)*			
ERMHAN [50]	✔	OWL DL & rule-based	✔		✔
PERSONA [51]	✔	rule-based & supervised learning			

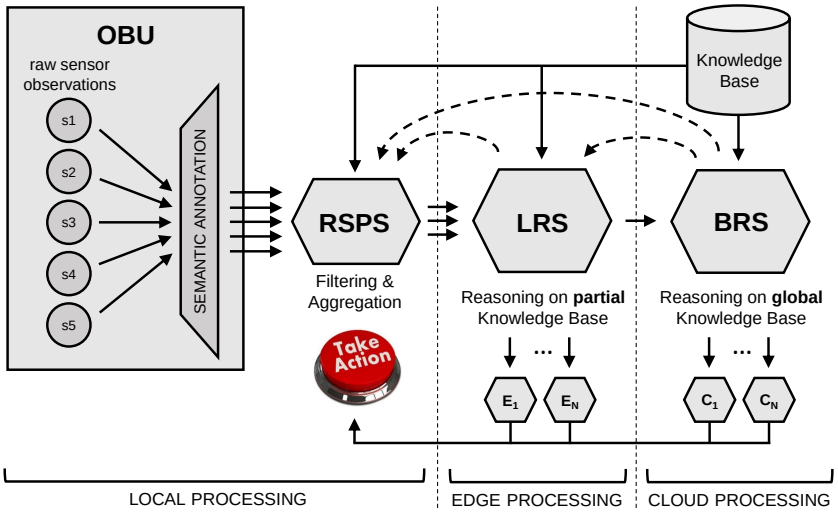


Figure 2.1: High-level architecture of the proposed cascading reasoning framework. The blocks represent the several components, the arrows indicate how the data flows through these components. The dotted arrows indicate possible feedback loops to preceding components.

The information in the knowledge base can be managed in a centralized database system, which can be an RDF triple store. In this case, the triple store contains mappings of the existing data architecture to the supported ontological formats. In addition, the information can also be managed in a regular database system that supports ontology-based data access (OBDA) [52].

The OBU refers to the infrastructure used to monitor the given environment. This infrastructure can consist of WSNs, BANs and other sensor platforms. The task of the OBU is threefold: (i) capture the raw sensor observations, (ii) semantically annotate these observations and (iii) push the resulting set of RDF triples on its corresponding output RDF data stream. This set of RDF triples should consist of a reference to the sensor producing the observation, the observed value and an associated timestamp.

The RSPS, LRS and BRS components are the stream processing and reasoning components. By configuration, only the relevant parts of the central knowledge base are available on each RSPS and LRS component, as these components typically do not need to know all domain knowledge and/or context information of the full system. On each BRS component, the full central knowledge base is available. Updates to the knowledge bases are coordinated from the BRS component(s).

The RSPS is situated locally and contains an RSP engine. The input of this engine consists of the RDF data streams produced by the OBU, or another RSPS. The task of the RSP engine is to perform some initial local processing of these input data streams, by aggregating and filtering the data according to its relevance. On the RSPS, little

to no reasoning is done on the data, depending on the use case. The output of the RSP engine is one or more RDF streams of much lower frequency, containing the interesting and relevant data that is considered for further processing and reasoning.

The LRS is situated in the edge of the network. Here, a local reasoner is running that is capable of performing more expressive reasoning, e.g., OWL 2 RL or OWL 2 DL reasoning. It takes as input the output RDF stream(s) of the RSPS, or another LRS. As the velocity of these streams is typically lower than the original stream, computation time of the reasoning can be reduced. The service has two main responsibilities. First, it can push reasoning results and/or data patterns to another LRS, or a BRS in the cloud, for further processing. Again, the output stream typically is of lower frequency than the input streams. Second, it can also push results to one or more other external components that are capable of performing some first local actions. This allows for fast intervention, before the results are further processed deeper in the network.

The BRS is situated in the cloud. It also consists of an expressive reasoner that has access to the full central knowledge base. Typically, a small number of BRS components exist in the system, compared to several LRS and even more RSPS components. The reasoning performed by the BRS can be much more complex, as it is working on data streams of much lower frequency compared to the local and edge components. This enables to derive and define intelligent and useful insights and actions. These insights and action commands can be forwarded to other BRS or external components, which can then act upon the received information or commands.

In some use cases, it might be useful to provide feedback to preceding components in the chain. This is possible in the current architecture, by the use of feedback loops. This feedback can be seen as messages, e.g., events or queries, in the opposite direction of the normal data flow.

When deploying the architecture, each component in the system should be configured. To this end, the observer concept is used: each component, including external components, can register itself as an observer to an output stream of another component. In this way, the system components can be linked in any possible way. Hence, using the generic architecture, an arbitrary network of components can be constructed and configured. It should be noted that this is a push-based architecture, where the outputs of each component are immediately pushed to the input stream(s) of its observers.

Note that this architecture assumes the following prerequisites: (i) the security of the architecture has been set up and ensured; (ii) no loss of connectivity to the cloud is assumed; (iii) each component runs on a node with at least 1 GB of memory resources; and (iv) each component is available at all times.

Figure 2.2 shows a potential deployment of this architecture in a hospital setting. In this example, there is one OBU and RSPS per patient, one LRS per room, one BRS per department, and another BRS for the full hospital.

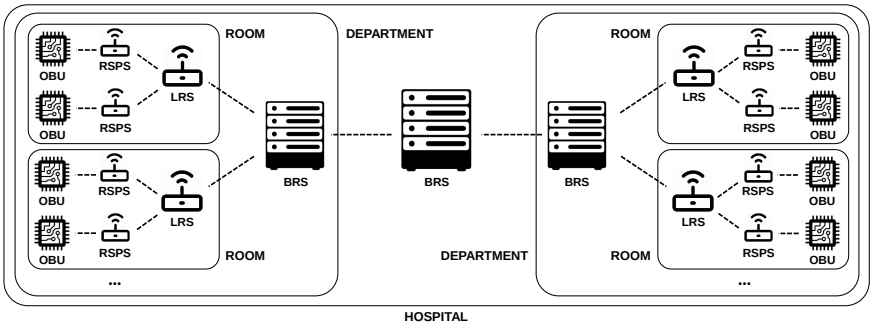


Figure 2.2: Potential deployment of the architecture of the cascading reasoning platform in a hospital setting. A network of components can be constructed. In this example, there is one Observation Unit (OBU) and RSP Service (RSPS) per patient, one Local Reasoning Service (LRS) per room, one Back-end Reasoning Service (BRS) per department, and another BRS for the full hospital. Potential external components are omitted from this figure.

2.4 Use case description and set-up

A PoC has been developed for a use case situated in pervasive healthcare. In this section, this use case is described in detail, followed by a discussion of how the use case has been mapped to the architecture described in Section 2.3. Moreover, the designed continuous care ontology and the data sources for the use case are discussed. The implementation of the different architecture components is given in Section 2.5.

2.4.1 Pervasive health use case description

Consider a use case where a patient Bob is suffering from a concussion and is therefore being hospitalized. The EHR of Bob states that he suffers from a concussion, meaning that direct exposure to light and sound must be avoided. Both Bob's EHR and this medical domain knowledge are available in the knowledge base. Based on this data, an acceptable level for both light intensity and sound level to which Bob may be exposed to (one of *none*, *low*, *moderate*, *normal* or *high*) can automatically be suggested and added to the knowledge base. These personalized levels can then be adjusted by a doctor or the nursing staff, if required. For Bob, the acceptable level of both light intensity and sound is *low*. For each property, a mapping between the acceptance levels and absolute threshold values is also part of the medical domain knowledge. For example, a *moderate* light intensity level is mapped to a light intensity threshold of 360 lumen, meaning that 360 lumen is the maximum light intensity that a patient with this acceptance level may be exposed to. In Bob's case, the threshold values for light intensity and sound are 180 lumen and 30 decibels, respectively.

The hospital room where patient Bob is accommodated, is equipped with an OBU. This OBU has multiple sensors, among which a light and sound sensor. The OBU

can also detect the presence of people in the room through the presence of a Bluetooth Low Energy (BLE) sensor (beacon). As all patients and nurses in the hospital are wearing a BLE bracelet containing a personal BLE tag, the system is able to discover all relevant people present in the room.

When the observed light intensity or sound values in Bob's room exceed the thresholds related to the acceptance levels found in his EHR, a *possibly unpleasant situation* for Bob is detected. This situation is called a *symptom*. Possibly unpleasant means that the situation should be further investigated. When the situation also is an actual *alarming situation* for patient Bob, it is called a *fault*. When a fault is detected, certain action(s) can/should be taken by the system to try to solve the fault. Whether or not a symptom implies a fault and thus requires action, depends on information regarding the actual context. For this use case, this is only true if the patient who is accommodated in the room where the threshold is crossed, is sensitive to the measured property, e.g., light intensity or sound.

Once a fault is detected, the system will try to solve it by taking one or more actions. An action can be static, or it can depend on other data. For this use case, the action taken depends on the presence of a nurse in the patient room at the time of the fault detection. When a nurse, who is responsible for that patient, is present in the room, the fault is considered less severe. In such a situation, it is likely to assume that the nurse knows how to treat the patient and that precautions have been taken to shield the patient from direct light and sound exposure. However, it might be useful to warn the nurse of the exceeded threshold by means of a message on an information display or on a mobile device. When no nurse is present in the room, the fault is much more severe. Actions should be taken to resolve the fault: a nurse should be called by the system to go on site and check the situation in the room. Awaiting the arrival of the nurse, local action can already be taken. For example, in case the fault is caused by a high light intensity observation, the light level can already be automatically reduced to a more acceptable level, e.g., by dimming the lights or closing the window blindings. Again, a warning can be displayed on an information display to make people in the room aware of what is happening.

2.4.2 Architectural use case set-up

To implement the use case, the architecture presented in Section 2.3 is used. An overview of the architectural set-up for this use case is shown in Figure 2.3.

To map the architecture, a few assumptions about the hospital and its rooms are made. The hospital consists of multiple departments. On each department, multiple nurses are working. In each department, several hospital rooms are located, both single-person and multi-person rooms. In each room, there exists exactly one OBU and RSPS per bed, i.e., if accommodated, per patient. There is one LRS per room, independent of the amount of patients. Over the full hospital, only one BRS exists.

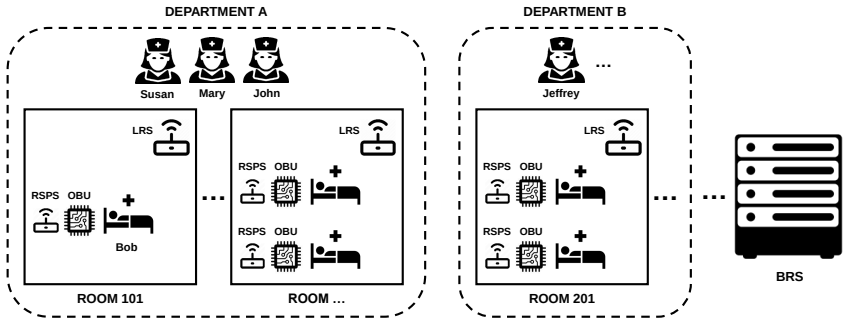


Figure 2.3: Architectural set-up for the Proof-of-Concept use case. Each hospital room contains one Local Reasoning Service (LRS), and one Observation Unit (OBU) and RSP Service (RSPS) per patient. There is only one Back-end Reasoning Service (BRS) in the hospital. Patient Bob is accommodated in room 101 of department A, which is supervised by nurses Susan, Mary and John.

For this implementation, such a simple set-up is considered. However, in a real-life set-up, there will typically be more than one BRS component in the system, e.g., an extra BRS per department, as indicated in Figure 2.2.

In each room, the OBU consists of a BLE sensor, and multiple environmental and/or body sensors. For the concussion diagnosis and corresponding sensitiveness to light intensity and sound, a light and sound sensor are sufficient. Of course, in a real-life use case, the available medical domain knowledge will consist of multiple diagnoses. Accordingly, the OBU will then also consist of potentially other sensors. For the system to work correctly, accuracy of the sensors is required. For example, the range of the BLE beacon should be correctly configured, such that it does not detect BLE devices that are nearby, but in another room.

2.4.3 Continuous care ontology

A continuous care ontology has been designed [53] to describe existing medical domain knowledge, to enable the semantic annotation of the sensor observations, and to allow modeling all available context information. For PoC purposes, a new ontology has been designed for this. However, to link it with existing medical ontologies, a mapping to the SNOMED ontology can be added to the ontology.

The starting point for this ontology was the ACCIO ontology [54]. This is an OWL 2 DL ontology designed to represent different aspects of patient care in continuous care settings [55]. It links to other existing ontologies, such as the SSN [9] and SAREF [56] ontologies. For this use case, the ACCIO ontology allows to represent hospital departments, rooms, sensors, BLE bracelets, observations, nurses, patients, actions, nurse calls, etc. These concepts can be represented, as well as the relations between them. Moreover, the ontology contains some concepts for this use case

that allow the inference of certain situations and hence easier query writing. An example is `NursePresentObservation`, which is defined as a BLE tag observation of a bracelet owned by a nurse:

```
NursePresentObservation ≡
    (hasResult some (
        observedDevice some (
        BLEBracelet and (deviceOwnedBy some (
                        Person and (hasRole some StaffMemberRole))))))
    and (madeBySensor some BLEBeacon).
```

The existing ACCIO ontology has been further extended for this work with some key concepts and relations specific for the current use case. This extension is the `CareRoomMonitoring.owl` ontology. Here, all possible medical symptoms, diagnoses, symptoms and faults are defined. For this use case, the `Concussion` class is defined as being equivalent to a `Diagnosis` that has two medical symptoms, being light sensitiveness and sound sensitiveness:

```
Concussion ⊑
    Diagnosis
    and (hasMedicalSymptom some SensitiveToLight)
    and (hasMedicalSymptom some SensitiveToSound).
```

Moreover, the `SoundAboveThresholdFault` class is defined as:

```
SoundAboveThresholdFault ≡
    (hasSymptom some SoundAboveThresholdSymptom)
    and (madeBySensor some (
            isSubsystemOf some (
                hasLocation some (
                    isLocationOf some (
                        (hasDiagnosis some (
                            hasMedicalSymptom some SensitiveToSound))
                        and (hasRole some PatientRole))))))).
```

Two conditions need to be fulfilled for an `Observation` individual to also be a `SoundAboveThresholdFault` individual. First, it needs to have a `SoundAboveThresholdSymptom`, indicating the possibly unpleasant situation. Second, the observation needs to be made by a sensor system situated at the same location as a sound sensitive patient, i.e., a patient with a diagnosis that implies sound sensitiveness. If that is also the case, the possibly unpleasant situation is alarming. For this use case, the definition of `LightIntensityAboveThresholdFault` is completely similar.

The approach of using medical symptoms, diagnoses, symptoms and faults allows complete separation of diagnosis registration and fault detection. For an observation to be a fault, the exact diagnosis of the patient located at the corresponding room is unimportant; only the medical symptom, e.g., sensitiveness to light or sound, needs to be known. Vice versa, a patient's sensitiveness to light and sound, or any other medical symptom, does not need to be explicitly registered in the system; registering the diagnosis is sufficient. Because the diagnosis is already defined

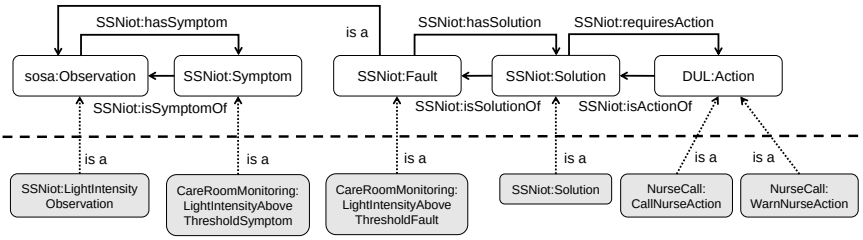


Figure 2.4: Overview of the most important ontology patterns and related classes of the Proof-of-Concept use case. The example for light intensity is given; the classes and patterns for sound are similar.

in the system according to medical domain knowledge, its corresponding medical symptoms are implicitly known.

By design, the ACCIO ontology contains different patterns to represent the logic related to this use case. For example, a `Fault` is an `Observation` that needs a `Solution` through `hasSolution`, and a `Solution` requires an `Action` via `requiresAction`. Each such `Action` has exactly one `Status` via `hasStatus`, indicating the status in the life cycle of the action. To this end, a general overview of the most important ontology patterns and classes is presented in Figure 2.4.

2.4.4 Data sources

As can be seen in Figure 2.1, each RSPS, LRS and BRS component of the system works with two data sources: the knowledge base and streaming data. Both are use case specific.

2.4.4.1 Knowledge base

For this use case, the knowledge base consists of the continuous care ontology, described in Section 2.4.3, and available context data. This information can be managed in a centralized database system, which can be an RDF triple store, or a database system that supports OBDA [52]. Examples of OBDA systems are Ontop [57] and Stardog [58]. By configuration, only the relevant parts of the knowledge base are available on each LRS and BRS. On the BRS, the full knowledge base is available.

Context data can be considered as static data: although it can change over time, the number of updates is low compared to the number of times a query evaluation is performed on the data before it changes. For this PoC, the context data includes information about the hospital layout, patients and nurses, the OBU and connected sensors, and BLE wearables. Changes to this context data are less frequent but do occur. For example, a newly diagnosed person is being accommodated in a room, or a new nurse starts working at a department. In these cases, the knowledge base of

each relevant component needs to be updated. This is coordinated from the central knowledge base at the BRS.

On each RSPS and LRS component, only the information about the current department and room is known. For the nurses, context data related to all hospital nurses of the own department is available on each RSPS and LRS. On each RSPS, only the patient information of its associated patient is available. Similarly, patient data of all patients in the room is present in the knowledge base of each LRS. Consequently, on the RSPS and LRS of Bob's room, four persons are defined: patient Bob, and three nurses, Susan, Mary and John. Bob is lying in room 101 of department A. According to the modeled diagnosis, he is suffering from a concussion. For both light intensity and sound, Bob's threshold values are modeled. Moreover, each person is assigned a BLE bracelet.

The OBU in room 101 is uniquely identifiable by its MAC address. Three sensors are connected to the OBU: a sound sensor, a light sensor and a BLE beacon. Each sensor has a unique identifier composed of the MAC address of the OBU and a sensor ID which is unique on a single OBU.

Moreover, a threshold value is modeled for the light and sound sensor. These thresholds are identical to the corresponding threshold values of Bob for exposure to light intensity and sound. Directly linking these thresholds to the sensors themselves is crucial for the filtering at the RSP engine, as will be explained in Section 2.5.2. The process of mapping the thresholds of a person, related to a received medical diagnosis, to thresholds of the sensors of the patient's OBU, needs to happen when a (new) diagnosis is made. This is achieved by running an appropriate query, inserting the sensor thresholds into the different knowledge bases.

Similarly to the patient data, only the associated OBU data is available in the knowledge base of an RSPS. On each LRS, all OBU data of the associated room is known.

2.4.4.2 Streaming data

For this use case, the streaming data is semantically annotated and pushed to different streams by an OBU. The semantic annotation is an important task. In this mapping process, the observations are modeled according to the continuous care ontology. Listing 2.1 shows the template of an RDF observation message for a sound observation: `(A)` denotes the sensor identifier, `(B)` the observation timestamp expressed in milliseconds, `(C)` the observation timestamp in `xsd:dateTime` format, `(D)` the observed value, and `(E)` the corresponding unit. The name of each observation is unique due to the observation identifier `(A)-(B)`. For a light intensity or BLE tag observation, the template is similar. In case of a BLE tag observation, the result contains the ID of the observed BLE device.

Listing 2.1: Template of a semantically annotated sound sensor observation

```
obs:Observation_(A)-(B) rdf:type sosa:Observation ;
    General:hasId [ General:hasID "(A)-(B)"^^xsd:string ] ;
    sosa:observedProperty [ rdf:type SSNiot:Sound ] ;
    sosa:madeBySensor [ General:hasId [ General:hasID "(A)"^^xsd:string ] ] ;
    sosa:resultTime "(C)"^^xsd:dateTime ;
    sosa:hasResult [ rdf:type SSNiot:QuantityObservationValue ;
        DUL:hasDataValue "(D)"^^xsd:float ;
        SSNiot:hasUnit "(E)"^^xsd:string ] .
```

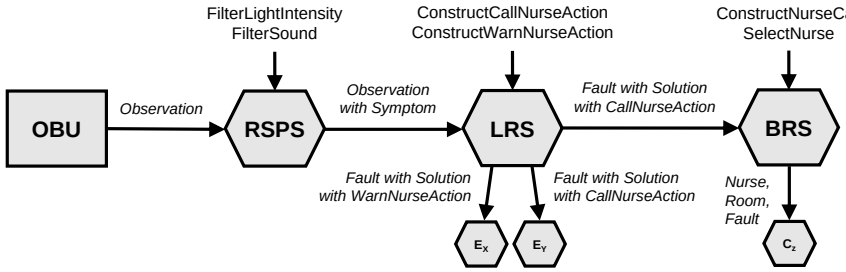


Figure 2.5: Overview of the components of the Proof-of-Concept. The inputs and outputs of each main component are shown in italic, as well as the queries executed on each RSP Service (RSPS), Local Reasoning Service (LRS) and Back-end Reasoning Service (BRS). The additional components E_X and E_Y represent the components taking local action. C_Z represents the component taking care of the actual nurse call. Feedback loops are omitted from the overview.

2.5 PoC component implementation

This section will go into detail on the implementation of the PoC, introduced in Section 2.4, on each of the four main architecture components of the cascading reasoning framework presented in Section 2.3. Figure 2.5 gives an overview of the main components of the PoC. It shows the inputs and outputs of each component, which are all events containing RDF triples, as well as the queries that are being executed on each RSPS, LRS and BRS component.

2.5.1 OBU

In every hospital room, one OBU per patient is present to monitor the environment. For this PoC, the single-person room of patient Bob is considered, where one OBU is installed. As explained in Section 2.4.4.1, this OBU consists of three sensors: a light sensor, a sound sensor and a BLE beacon. In a realistic system, the light and sound sensors can be part of a sensor board, such as a GrovePi+. As a BLE beacon is a different type of sensor, it is not part of this board. Therefore, for the implementation of this PoC, the OBU pushes the sensor observations in two separate RDF data streams: one stream (http://rspc1.intec.ugent.be/grove) containing the sensor board

observations, and one stream (http://rspc1.intec.ugent.be/ble) containing the BLE tag observations. Recall that the architecture is push-based, i.e., each observation made by the sensors is immediately pushed on these streams.

2.5.2 RSPS

As explained in Section 2.3, the RSPS is situated locally, and consists of an RSP engine. In the given use case, locally means within the hospital room. In concrete, there is one RSPS component per OBU, i.e., per patient. Therefore, for this PoC, one RSPS component is deployed in Bob's room.

For the RSPS, the used RSP engine is C-SPARQL [21], because of its support for static context data [20]. Both input and output of the RSP engine are RDF streams. It is used together with the RSP Service Interface for C-SPARQL [59], which offers a simple RESTful interface to interact with C-SPARQL. In terms of reasoning capabilities, C-SPARQL incorporates the possibility to perform RDFS reasoning. However, reasoning is time-consuming and may take too much time when fast reevaluation of the continuous queries is required. Therefore, for this use case, no reasoning is performed by C-SPARQL.

In the cascading reasoning approach, RSP engines are used to filter the change frequency of the data streams. Only interesting information is retained. Therefore, appropriate continuous queries are constructed that intelligently aggregate and filter the data streams for the use case at hand.

For this use case, two similar C-SPARQL queries are running on the RSPS component: `FilterSound` and `FilterLightIntensity`. As explained before, a window needs to be placed on top of the continuous data streams. For these queries, a logical window is used, which is a window extracting all triples occurring during a certain time interval. In concrete, both queries are executed every 5 s, on a logical sliding window containing all light, sound and BLE tag observations of the previous 6 s. The window size of 6 s is chosen as such to avoid situations where certain observations fall between two windows. Theoretically, this should not be possible when the window size and sliding step are both equal, but, in practice, a real implementation of the system may exhibit a lag of a few milliseconds between two query executions. The triples constructed by the query are sent as RDF messages to the event stream of the LRS. Listing 2.2 shows the `FilterSound` query, which is discussed in the next paragraphs. The `FilterLightIntensity` query and its motivation are completely similar.

Lines 14–16 of the `FilterSound` query define its inputs: the stream with sensor board observations, the stream with BLE tag observations, and the context data available in the local knowledge base.

The `WHERE` clause of the query consists of two large parts (lines 18–44 and lines 46–59). Considering the first part, its first section (lines 19–25) extracts any sound sensor observation in the input window. The second section (lines 27–41) contains

an optional pattern that looks at BLE tag observations in the window corresponding to the BLE bracelet of a nurse. Note that this pattern only matches BLE tag observations of nurses that are working on the department the hospital room belongs to. This is because context information about the BLE bracelets of other nurses is not available in the local knowledge base. The rationale for this is that it is assumed that nurses of other departments know too little about the patients of the current department. Hence, their presence in the room should not affect the outcome of the query.

Line 43 of the query contains an important filter. It checks for each sound observation whether the observed sound value is higher than the threshold of the sound sensor that measured the value. As explained in Section 2.4.4.1, this sensor threshold exactly corresponds to the patient's sound exposure threshold defined in the patient's EHR. Only if this threshold is crossed, the observation is retained, as this might imply a possibly unpleasant situation, i.e., a symptom.

The second part of the `WHERE` clause consists of a second filter clause (lines 46–59). This filter will ensure that the `WHERE` clause will only yield a result pattern, if there currently is a BLE tag observation present in the input window, which corresponds to a BLE bracelet of a patient. If this is not the case, no relevant patient is currently present in the hospital room. This means that the crossed sound threshold is not a problem. Note that, if the presence of a patient is detected, this automatically is the patient lying in the bed corresponding to the RSPS. This is again a consequence of the fact that the local knowledge base of the RSPS only contains patient information about its corresponding bed.

Lines 61–62 of the query sort the results of the `WHERE` clause according to the observation timestamp in descending order, and only retain the first result. In this way, if there are multiple sound observations above the sensor threshold, only the most recently observed sound value is retained in the query result.

If the query yields a result, new triple patterns are constructed by the `CONSTRUCT` part (lines 2–12). The high sound observation, which crossed its sensor's threshold, will be given a newly created `SoundAboveThresholdSymptom`. Other information on the particular observation is copied as well, together with any BLE tag observation of a nurse's bracelet, detected through the `OPTIONAL` clause.

2.5.3 LRS

The LRS is situated in the edge of the network, as discussed in Section 2.3. In this use case, this again corresponds to the hospital room. Therefore, in this PoC, there is one LRS per hospital room.

2.5.3.1 Reasoning service

To implement the LRS, and also the BRS addressed in Section 2.5.4, a reasoning service component is implemented. In this reasoning service, the OWL 2 RL reasoner

Listing 2.2: `FilterSound` query running on the RSP Service (RSPS) component

```
CONSTRUCT {
    _:sym rdf:type CareRoomMonitoring:SoundAboveThresholdSymptom ;
        General:hasId [ General:hasID ?o_id ] .
    ?m_o SSNiot:hasSymptom _:sym ; rdf:type sosa:Observation ;
        General:hasId [ General:hasID ?o_id ] ; sosa:hasResult ?m_r .
    ?m_r DUL:hasDataValue ?m_v ; SSNiot:hasUnit ?u .
    ?m_o sosa:resultTime ?m_t ; sosa:madeBySensor ?s .

    ?ble_ob1 rdf:type sosa:Observation ; sosa:madeBySensor ?ble_s1 ;
        sosa:resultTime ?ble_ob1_time ; sosa:hasResult ?ble_r1 .
    ?ble_r1 rdf:type SSNiot:TagObservationValue ;
        SAREFiot:observedDevice ?ble_b1_id .
}
FROM STREAM <http://rspc1.intec.ugent.be/grove> [RANGE 6s STEP 5s]
FROM STREAM <http://rspc1.intec.ugent.be/ble> [RANGE 6s STEP 5s]
FROM <http://localhost:8181/context.ttl>
WHERE {
    {
        ?m_o rdf:type sosa:Observation ; sosa:hasResult ?m_r ;
            sosa:madeBySensor [ General:hasId [ General:hasID ?s_id ] ] ;
            General:hasId ?o_id_ent ; sosa:resultTime ?m_t .
        ?o_id_ent General:hasID ?o_id . ?m_r DUL:hasDataValue ?m_v .
        OPTIONAL { ?m_r SSNiot:hasUnit ?u } .
        ?s rdf:type SSNiot:LightSensor; General:hasId [ General:hasID ?s_id ] ;
            SSNiot:hasThreshold [ DUL:hasDataValue ?th ] .

        OPTIONAL {
            ?ble_ob1 rdf:type sosa:Observation ;
                sosa:madeBySensor [ General:hasId [
                    General:hasID ?ble_s1_id ] ] ;
                sosa:resultTime ?ble_ob1_time ; sosa:hasResult ?ble_r1 .
            ?ble_r1 rdf:type SSNiot:TagObservationValue ;
                SAREFiot:observedDevice [ General:hasId [
                    General:hasID ?ble_b1_id ] ] .
            ?ble_s1 rdf:type SSNiot:BLEBeacon ;
                General:hasId [ General:hasID ?ble_s1_id ] .
            ?ble_b1 rdf:type SAREFiot:BLEBracelet ;
                General:hasId [ General:hasID ?ble_b1_id ] .
            ?p1 rdf:type DUL:Person ; SAREFiot:ownsDevice ?ble_b1 ;
                DUL:hasRole [ rdf:type RoleCompetenceAccio:StaffMemberRole ]
        }

        FILTER (xsd:float(?m_v) > xsd:float(?th))
    }

    FILTER (EXISTS {
        ?ble_ob2 rdf:type sosa:Observation ; sosa:hasResult ?ble_r2 ;
            sosa:madeBySensor [ General:hasId [
                General:hasID ?ble_s2_id ] ] .
        ?ble_r2 rdf:type SSNiot:TagObservationValue ;
            SAREFiot:observedDevice [ General:hasId [
                General:hasID ?ble_b2_id ] ] .
        ?ble_s2 rdf:type SSNiot:BLEBeacon ;
            General:hasId [ General:hasID ?ble_s2_id ] .
        ?ble_b2 rdf:type SAREFiot:BLEBracelet ;
            General:hasId [ General:hasID ?ble_b2_id ] .
        ?p2 rdf:type DUL:Person ; SAREFiot:ownsDevice ?ble_b2 ;
            DUL:hasRole [ rdf:type RoleCompetenceAccio:PatientRole ] .
    })
}
ORDER BY DESC(?m_t)
LIMIT 1
```

RDFox [14] is used. It was chosen over other, more expressive reasoners such as HermiT because of its highly efficient reasoning [14].

The continuous care ontology presented in Section 2.4.3 is an OWL 2 DL ontology, whereas RDFox is an OWL 2 RL reasoner. The OWL 2 RL profile is designed to implement reasoning systems using rule-based reasoning engines [60]. Therefore, certain restrictions are present. One restriction is that an OWL 2 RL reasoner cannot infer the existence of individuals that are not explicitly present in the knowledge base. For example, according to the OWL 2 DL definition of `Concussion` presented in Section 2.4.3, for each `Concussion` individual `p`, an OWL 2 DL reasoner will create new triples `p rdf:type Diagnosis`, `p hasMedicalSymptom [ rdf:type SensitiveToLight ]` and `p hasMedicalSymptom [ rdf:type SensitiveToSound ]`. An OWL 2 RL reasoner cannot infer the second and third triple, which forms a problem, as their existence is used in the `SoundAboveThresholdFault` definition. Therefore, such triples are explicitly added to the ontology for a single `Concussion` individual, which is then used to model the diagnosis of a person.

By configuration, the reasoning service has a number of predefined event-based processing SPARQL queries. `SELECT`, `CONSTRUCT` and `UPDATE` queries are supported. Importantly, these queries can be ordered in the component's configuration file. This is required because they can depend on each other: an `UPDATE` query may for example work on new triples constructed by a `CONSTRUCT` query.

For a `CONSTRUCT` and `SELECT` query, one or more observers can be defined to which the resulting triples need to be sent. These observers are endpoints, such as streams. For example, in the LRS, the triples constructed by a `CONSTRUCT` query can be sent to the event stream of the BRS. Variable bindings outputted by `SELECT` queries can also be sent to external components for further processing. For a `CONSTRUCT` query, it can also be defined whether the constructed triples should be added to the local triple store. `UPDATE` queries only update the triples in the local triple store. After every addition or removal of triples to/from the triple store, incremental OWL 2 RL reasoning is performed. Incremental reasoning is a technique where the implicit assertions in the knowledge base are incrementally maintained, to avoid recomputing all assertions each time a new set of triples is added [14]. This technique is often employed by semantic query and reasoning engines to improve reasoning performance.

In Figure 2.6, an overview of the functionality of the reasoning service is given. The main running thread of the reasoning service component works with a single queue of incoming data events and feedback queries. This means that data events and feedback queries can be sent to the system, where they are added to the queue and sequentially processed in order of arrival.

In this use case, a data event is an RDF message that is arriving from an RSPS, LRS or BRS component. When the event is removed from the queue by the main running thread, i.e., when the processing starts, the event triples are temporarily added to the triple store. Next, incremental OWL 2 RL reasoning is performed, generating

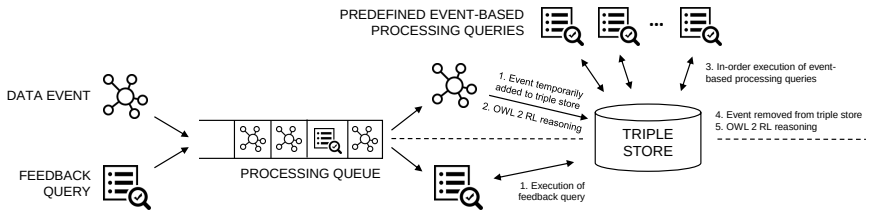


Figure 2.6: Overview of the functionality of the implemented reasoning service, used by the Local Reasoning Service (LRS) and Back-end Reasoning Service (BRS)

all inferred triples. Each predefined query is then sequentially executed in the defined order. When finished, the event triples are again removed from the triple store, after which a final incremental reasoning step is executed.

Besides the event-based processing queries that are executed on every event, in some cases, it might be interesting to execute a specific query once on the triples of the triple store, e.g., when the status of an action needs to be updated. Such feedback queries can also be sent to the system. When a feedback query is processed by the reasoning service, it is simply executed on the local triple store. Because these feedback queries are straightforward for this use case, they are not further discussed.

2.5.3.2 Event-based processing queries

For this use case, incoming events at the LRS are RDF messages containing the triples constructed by the `FilterSound` (Listing 2.2) and `FilterLightIntensity` queries running on the RSPS. These events contain light and sound observations that are possibly unpleasant, i.e., that have a symptom. The following queries are sequentially run on the LRS when an event is being processed:

- `ConstructCallNurseAction` (Listing 2.3): This `CONSTRUCT` query looks for an instance of a `DetectedFault` (lines 12–14), i.e., it analyzes whether any fault is detected by the system. Recall that an observation is a fault if given conditions are fulfilled. For each fault, these conditions are integrated in the specification of the fault in the ontology. In this way, when the conditions are fulfilled, the fault has been inferred by the reasoner after the addition of the event triples.

  If a fault is detected – assuming the three filter clauses, addressed in the following paragraph, are passed – new triples are constructed (lines 2–8). A solution for the fault is created. Each solution requires a corresponding action, in this case a `CallNurseAction`. This action implies that a nurse should be called to go on site to check the room. A `CallNurseAction` individual has four statuses in its life cycle: `New`, when the action is created but not handled yet; `Active`, when a component has activated the action and is starting the

nurse assignment process; `Assigned`, when a nurse has been assigned; and `Finished`, when the nurse has arrived at the room.

Three `FILTER NOT EXISTS` patterns are added to the query's `WHERE` clause. Only if they are passed, a solution is created. The first filter (line 17) ensures that no `NursePresentObservation` is present. Recall from Section 2.4.3 that this is a BLE tag observation that is associated with the BLE bracelet of a nurse. If this information is available in the event, this means that the RSPS has detected the presence of a nurse in the room. Obviously, no nurse should be called in this case; another action is then taken by the `ConstructWarnNurseAction` query. The second filter (lines 19–22) ensures that not more than one solution is created for a fault, i.e., an observation, with the same ID. The third filter (lines 24–32) clause avoids that a `CallNurseAction` is created when there is already another `CallNurseAction` for that room that is not finished yet. In that case, it makes no sense to call a nurse for the second time. Note that this only holds if the unfinished `CallNurseAction` corresponds to a fault of the same type. The rationale for this is that a new call to the same room, but, for another fault, gives new information to a nurse that he/she can for example take into account when deciding how urgent the call is. Note that the concept `NewOrActiveOrAssignedAction` is used for this filter. This concept is defined as a subclass of `Action` that has one of these three statuses.

The triples constructed by the `ConstructCallNurseAction` are sent to the event stream of the BRS because this component will decide how to handle the `CallNurseAction`. To keep track of the status of the action, and to perform collect filtering in later executions of the query, the triples are also added to the local triple store.

- `ConstructWarnNurseAction` (Listing 2.4): This `CONSTRUCT` query is complementary to the `ConstructCallNurseAction` query in Listing 2.3. Hence, most parts are very similar.

  In the `CONSTRUCT` part of the query (lines 2–8), a `WarnNurseAction` is created instead of a `CallNurseAction`. In other words, a specific nurse `p1` should be warned, instead of calling a nurse to the room. This nurse will be present in the room at the time of the threshold crossing.

  To only construct this particular solution when a nurse is present, the `WHERE` class of the query is extended with the pattern in lines 16–19. This extension ensures that a `NursePresentObservation` is present, and retrieves the associated nurse. The two filters of the query are similar to the second and third filter of the `CallNurseAction` query. The second filter (lines 27–36) ensures that no new `WarnNurseAction` is created for the present nurse if she is currently being warned already.

Listing 2.3: `ConstructCallNurseAction` query running on the Local Reasoning Service (LRS) component

```
CONSTRUCT {
    _:f rdf:type ?t1 ; General:hasId [ General:hasID ?id ] ;
        sosa:madeBySensor ?sen ;
        SSNiot:hasSolution [
            rdf:type SSNiot:Solution ;
            SSNiot:requiresAction [
                rdf:type NurseCall:CallNurseAction ;
                General:hasStatus TaskAccio:New ] ] .
}
WHERE {
    {
        ?f1 rdf:type ?t1 ; General:hasId ?idobj ; sosa:madeBySensor ?sen .
        ?idobj General:hasID ?id . ?sen SSNiot:isSubsystemOf ?sys .
        ?t1 rdfs:subClassOf SSNiot:DetectedFault .
    }

    FILTER NOT EXISTS { ?ble_ob1 rdf:type NurseCall:NursePresentObservation . }

    FILTER NOT EXISTS {
        ?f3 SSNiot:hasSolution ?s1 ; General:hasId ?f3_idobj .
        ?f3_idobj General:hasID ?id .
    }

    FILTER NOT EXISTS {
        {
            ?f2 rdf:type ?t1 ; SSNiot:hasSolution ?s2 .
            ?s2 SSNiot:requiresAction ?a2 .
            ?a2 rdf:type NurseCall:CallNurseAction,
                         TaskAccio:NewOrActiveOrAssignedAction .
        }
        FILTER (?f1 != ?f2)
    }
}
```

The triples constructed by this query are not sent to the BRS because each `WarnNurseAction` is completely handled locally. However, the triples are again added to the local triple store.

2.5.3.3 Taking local action

The `ConstructWarnNurseAction` query running on the LRS creates a `WarnNurseAction` when a nurse is in the room and should be warned of a fault. To do so, an external component should be involved that is capable of communicating with the wearables, i.e., that can transfer the triples constructed by the query into a real notification on the nurse's wearable. This component should therefore be registered as observer of the `ConstructWarnNurseAction` query. It is also the task of this component to correctly update the status of the `WarnNurseAction` individual. This can be done by sending appropriate `UPDATE` queries to the LRS to delete and insert the correct triples in the local knowledge base. In Figure 2.5, this component is shown as component E_X.

Listing 2.4: `ConstructWarnNurseAction` query running on the Local Reasoning Service (LRS) component

```
CONSTRUCT {
    _:f rdf:type ?t1 ; General:hasId [ General:hasID ?id ] ;
        sosa:madeBySensor ?sen ;
        SSNiot:hasSolution [
            rdf:type SSNiot:Solution ;
            SSNiot:requiresAction [
                rdf:type NurseCall:WarnNurseAction ;
                DUL:hasContext ?p1 ; General:hasStatus TaskAccio:New ] ] .
}
WHERE {
    {
        ?f1 rdf:type ?t1 ; General:hasId ?idobj ; sosa:madeBySensor ?sen .
        ?idobj General:hasID ?id . ?sen SSNiot:isSubsystemOf ?sys .
        ?t1 rdfs:subClassOf SSNiot:DetectedFault .

        ?ble_ob1 rdf:type NurseCall:NursePresentObservation ;
            sosa:hasResult ?ble_r1 .
        ?ble_r1 SAREFiot:observedDevice ?ble_b1 .
        ?p1 SAREFiot:ownsDevice ?ble_b1 .
    }

    FILTER NOT EXISTS {
        ?f3 SSNiot:hasSolution ?s1 ; General:hasId ?f3_idobj .
        ?f3_idobj General:hasID ?id .
    }

    FILTER NOT EXISTS {
        {
            ?f2 rdf:type ?t1 ; SSNiot:hasSolution ?s2 .
            ?s2 SSNiot:requiresAction ?a2 .
            ?a2 rdf:type NurseCall:WarnNurseAction,
                        TaskAccio:NewOrActiveOrAssignedAction ;
                DUL:hasContext ?p1
        }
        FILTER (?f1 != ?f2)
    }
}
```

An example of a system that could be used for this additional component is DYAMAND [61]. This is a framework that acts as a middleware layer between applications and discoverable or controllable devices. It aims to provide the necessary tools to overcome the interoperability gaps that exist between devices from the same and other domains.

For the `ConstructCallNurseAction` query, the situation is different compared to the `ConstructWarnNurseAction` query. Constructed `CallNurseAction` individuals are sent to the BRS, where they are handled further. This means that the BRS, or another component interacting with the BRS, is responsible for updating the status of each `CallNurseAction` in the local knowledge base. Nevertheless, this does not mean that the LRS cannot take local action. Again, a component such as DYAMAND can be registered as observer to the `ConstructCallNurseAction` query. Based on the fault associated to an incoming `CallNurseAction`, it can already take some first local action. For example, in case of a `LightIntensityAboveThresholdFault`, the component can check whether the lights are switched off and the window blindings are closed. If not, this can then automatically be done. In Figure 2.5, this component is shown as component E_Y.

2.5.4 BRS

As addressed in Section 2.3, the BRS is running in the cloud of the network. Only one BRS is available for the entire hospital. The BRS has access to the full central knowledge base.

In the BRS, an ontology-based nurse call system (oNCS), similar to the one presented by Ongenae et al. [62], should be deployed. This includes complex probabilistic reasoning algorithms that determine the priority of a nurse call, based on the risk factors of the patient.

For this PoC, an incoming event on the BRS is the output of the `ConstructCallNurseAction` query. The following queries are sequentially run each time an event is processed:

- `ConstructNurseCall` (Listing 2.5): This `CONSTRUCT` query handles any `CallNurseAction` with status `New` that has been created as solution for a detected fault. It transforms the scheduled action into an actual nurse call. The query extracts the sensor system that caused the fault, which is set to have made the call. The reason for the call is the fault that the solution and action were created for. The triples constructed by this query are only added to the local triple store.

- `SelectNurse` (not listed): This query represents the complex nurse assignment process of the oNCS. For every nurse call constructed by the previous query, it should assign a nurse to the call. Again, a separate component—observing the results of this query, and using a system such as DYAMAND—can be used to

Listing 2.5: `ConstructNurseCall` query running on the Back-end Reasoning Service (BRS) component

```
CONSTRUCT {
    _:c rdf:type NurseCall:ContextCall ;
        NurseCall:callMadeBy ?sys ; General:hasStatus TaskAccio:New ;
        General:hasId [ General:hasID ?f_id ] ;
        TaskAccio:hasReason [ rdf:type ?t ] .
}
WHERE {
    ?f rdf:type ?t ; General:hasId ?f_idobj ;
        SSNiot:hasSolution ?sol ; sosa:madeBySensor ?s .
    ?t rdfs:subClassOf SSNiot:DetectedFault . ?s SSNiot:isSubsystemOf ?sys .
    ?f_idobj General:hasID ?f_id . ?sol SSNiot:requiresAction ?act .
    ?act rdf:type NurseCall:CallNurseAction ; General:hasStatus TaskAccio:New .
}
```

actually show the call to the nurse on his/her wearable. Important information for the nurse includes the room, patient and reason for the call, i.e., the fault. In Figure 2.5, this additional component is shown as component C_Z.

Note that the BRS is also responsible for keeping the status of each `CallNurseAction` and `ContextCall` at the corresponding LRS up to date.

2.5.5 Platform configuration

To adopt the implementation of the cascading reasoning framework for a specific use case, a number of things should be configured, like it has been described for the PoC use case. First, the network of components should be defined. The domain ontologies and context information should be described, and it should be configured which parts are relevant for which components. Moreover, the OBU should be configured to map its observations to RDF messages. Furthermore, for each RSPS component, the continuous queries should be defined. Similarly, appropriate event-based processing queries should be configured for the LRS and BRS service. For each component, its inputs should be defined by selecting the queries of other components to observe, or the output stream of an OBU. Potential use case specific feedback loops and external components to perform any actions should additionally be added. By configuring the aforementioned things, the implementation of the framework can be applied to other use cases with minimal effort.

2.6 Evaluation set-up

To demonstrate the functionality and evaluate the performance of the presented cascading reasoning framework, evaluation scenarios have been executed on the PoC implementation presented in Sections 2.4 and 2.5. Instead of using real sensors configured by the OBU, realistic scenarios have been created and simulated.

2.6.1 Evaluation goals

The first goal of the evaluation is to verify the correct working of the system in multiple scenarios. In particular, it is interesting to look how the system behaves in three distinct cases: one case where no fault occurs, one case where a fault is handled locally, and one case where a fault is handled globally.

The second evaluation goal is to verify the global performance of the cascading reasoning platform. According to legal stipulations in some countries, a nurse should have arrived at the correct location at most five minutes after a fault has occurred. Given this condition, according to the manufacturer cooperating with the research group, i.e., Televic NV (Izegem, Belgium), each nurse call assignment should be completed within 5 s after the fault occurrence, i.e., after the observation that crossed the threshold. This leaves ample time for the nurse to move to the room after receiving the alert. Note that, by definition, the actual call of the assigned nurse, which needs to be performed by an external component, is not part of the assignment process. This process finishes when the decision on which nurse to call is made.

Related to the second goal, the third evaluation goal is to get insight into the component-level performance of the cascading reasoning platform. This includes looking at the processing time and latency on each component, network latency, etc.

2.6.2 Evaluation scenarios

As indicated in the evaluation goals, three distinct scenarios are evaluated. Each scenario considers the use case and implementation of the PoC described in Sections 2.4 and 2.5. For these scenarios, a few assumptions are made:

- The evaluation scenarios consider the architectural use case set-up presented in Figure 2.3. They only consider room 101 of department A, where patient Bob is accommodated. Bob is diagnosed with a concussion.

- The OBU in Bob's room consists of exactly three sensors: a light sensor, a sound sensor, and a BLE beacon. The light sensor and sound sensor are producing one observation every second. The BLE sensor produces one or more observations every 5 s.

- All nurses of department A have a work shift comprising the whole period of the scenario.

- The domain knowledge in the knowledge base of each component consists of `CareRoomMonitoring.owl` and `NurseCall.owl` of the designed continuous care ontology. At the time of writing, the medical domain knowledge in `CareRoomMonitoring.owl` only contains one diagnosis, being concussion, and two medical symptoms, being sensitiveness to light and sound.

- The static context data in the local knowledge base of the RSPS and LRS exactly matches the context data described in Section 2.4.4.1. On the BRS, the knowledge base also contains similar context data about other rooms and patients. For this evaluation, it is assumed the hospital has two departments with 10 single-person rooms each. In each room, one OBU is present containing a light, sound and BLE sensor. Each room contains one diagnosed patient. On each department, three nurses are working during the current shift.

- On each component, the queries as given in Section 2.5 are running. To focus on the evaluation of the cascading platform, and not on the specific nurse call algorithm used, no full-fledged oNCS implementation is used on the BRS. Instead, a more simple version of the `SelectNurse` query is running, selecting one nurse of the department to assign to each nurse call.

- During the scenario runs, no background knowledge or context data updates are sent out from the BRS to the local knowledge bases.

- No real external components are actively present in the evaluation architecture. This is purely for evaluation purposes, as the goal is to look at the main components. As a consequence, no actual local actions are taken by real components. However, one mock-up external component exists, which is registered as an observer to the `ConstructWarnNurseAction` query on the LRS and the `SelectNurse` query on the BRS. In this way, the reasoning services send outgoing events (query results) to this component, allowing for calculating the latency of the components.

- Action statuses are also not automatically updated by feedback loops. Instead, the evaluation scripts ensure that, after each scenario run, the components are restarted, resetting the knowledge bases of all components to the pre-scenario state.

- During the scenario runs, all nodes are fully available. No loss of connectivity to the cloud occurs.

For all three scenarios, the preconditions are the same. Patient Bob is the only person present in hospital room 101. As Bob is recovering from a concussion, measures have been taken to protect him from direct exposure to light and sound. Therefore, the lights are dimmed, the window blindings are closed, and the television in the room is switched off. The light intensity is stable at 125 lumen, and the sound is stable at 10 decibels.

Baseline scenario During this scenario, the light intensity and sound values remain stable at 125 lumen and 10 decibels, respectively. No threshold is crossed, so that no symptom is created or fault is inferred. Hence, no nurse should be warned or called.

In both scenarios 1 and 2, assume Bob's wife enters the room after 30 s. Upon entering the room, she turns on the lights. As a result of turning on the lights, observed light intensity values rise up to 400 lumen.

Scenario 1 Assume nurse Susan is present in Bob's room at that moment. Normally, Susan should know Bob's diagnosis, and hence directly tell Bob's wife to turn off the lights again. However, assume the system will notify Susan on her bracelet that Bob is sensitive to light and that there is currently too much light intensity exposure for him. This happens at the LRS through an extra component, as explained in Section 2.5.3. As a consequence, after 10 s, the lights are turned off again, and the observed light intensity values decrease to 125 lumen. The BRS is not involved in this scenario, as no nurse should be called. After 20 more seconds, Susan leaves the room, and the scenario ends.

Scenario 2 Assume no nurse is present in Bob's room when his wife enters and turns on the lights. Hence, a nurse call will be initiated by the system, involving the RSPS, LRS and BRS components. In the BRS, nurse Susan is selected (**scenario 2a**). Thirty seconds later, Susan arrives at the room, explains to Bob's wife that she should not turn on the lights, and turns them off. Hence, five seconds after Susan's arrival, the observed light intensity values decrease to 125 lumen. Within this short time frame, Susan will receive another warning on her bracelet, triggered by the LRS (**scenario 2b**). Again, Susan stays for 20 s in the room. Afterwards, she leaves the room, and the scenario ends.

In Table 2.3, an overview is given of the observed values of the light sensor and the BLE sensor, for both scenarios 1 and 2.

Each scenario has been simulated 50 times. For each scenario, Gaussian noise with a variance of 1 has been added to the light intensity and sound observations. For all communication between components, the HTTP protocol is used. Hence, pushing an observation or query result on a component's stream means sending an HTTP POST request to the stream endpoint containing the data in the request body. The following metrics are calculated:

- **Network latency** of an event X from A to B: This is the time between the outgoing event X at component A and the incoming event X at component B.
- **RSPS processing time** and **RSPS latency**: Processing on the RSPS is not event-based, but window-based, i.e., time-based for the `FilterSound` and `FilterLightIntensity` queries. Say an observation event X arrives at the RSPS at time t_X. If this observation leads to a symptom, one of both queries will construct an output containing a symptom for observation X. Say t_Y is the time at which this RSPS query output is sent to its observers. The RSPS processing

Table 2.3: Overview of the observed values of the light sensor and the BLE sensor at each timestamp, for scenarios 1 and 2. The timestamps are in seconds since the start of the scenario, the light values in lumen, and for the BLE values, 'B' represents an observation of the BLE bracelet assigned to Bob, and 'S' similarly for Susan's bracelet. Changes to observed values only occur on multiples of 5 s. Note, however, that the light sensor samples every second, and the BLE sensor every 5 s.

(a) Scenario 1

Time	Light	BLE
0	125	B,S
5	125	B,S
10	125	B,S
15	125	B,S
20	125	B,S
25	125	B,S
30	400	B,S
35	400	B,S
40	125	B,S
45	125	B,S
50	125	B,S
55	125	B,S

(b) Scenario 2

Time	Light	BLE
0	125	B
5	125	B
10	125	B
15	125	B
20	125	B
25	125	B
30	400	B
35	400	B
40	400	B
45	400	B
50	400	B
55	400	B
60	400	B,S
65	125	B,S
70	125	B,S
75	125	B,S

time for that observation is then defined as $t_Y - t_X$. By definition, the RSPS latency equals the RSPS processing time. Note that, by definition, the RSPS processing time is not defined for observations that do not lead to a symptom.

- **LRS processing time**, **LRS latency** and **LRS queuing time**: Processing on the LRS is event-based. When an event arrives at t_{in}, the event is put in the queue. After the queuing time, the processing of the event starts at t_{start}. The event is added to the knowledge base, and queries are executed. An event can only lead to either a `CallNurseAction` or `WarnNurseAction`, but not both, i.e., only one LRS query can construct a result. If this is the case, say t_{out} is the time at which this outgoing event is sent to its observers. After all queries are executed, the event is removed from the knowledge base and processing ends at t_{end}. Given these definitions, the LRS processing time is defined as $t_{end} - t_{start}$; the LRS latency, if an outgoing event is sent, as $t_{out} - t_{in}$; and the LRS queuing time as $t_{start} - t_{in}$.

- **BRS processing time**, **BRS latency** and **BRS queuing time**: Definitions are similar to the LRS case. The outgoing event corresponding to the BRS latency always contains the nurse that is assigned to the nurse call by the `SelectNurse` query.

- **Total system latency**: The total system latency is defined as the total time that an observation is in the system. For an observation that generates a `WarnNurseAction`, this is the time until the `WarnNurseAction` is created and can be sent to an external component. Hence, it is the sum of the OBU-RSPS network latency, RSPS latency, RSPS-LRS network latency and LRS latency. For an observation that generates a `CallNurseAction`, this is the time until a nurse is assigned on the BRS. Hence, the system latency is the sum of of the OBU-RSPS network latency, RSPS latency, RSPS-LRS network latency, LRS latency, LRS-BRS network latency and BRS latency.

2.6.3 Hardware characteristics

To perform the evaluation, the architecture components in the evaluation set-up have been implemented as Docker containers on real hardware components. Hardware characteristics of the device running the OBU container are omitted because the observations are simulated. Hence, results do not depend on the performance of this component; the OBU just sends the observations to the RSPS streams. To host the RSPS, an Intel NUC, model D54250WYKH, is used. This device has a 1300 MHz dual-core Intel Core i5-4250U CPU (turbo frequency 2600 MHz) and 8 GB DDR3 1600 MHz RAM [63]. The edge component hosting the LRS also is an Intel NUC with the same model number and specifications. In the cloud, the BRS is hosted by

Table 2.4: Average amount of incoming RDF events on the RSP Service (RSPS), Local Reasoning Service (LRS) and Back-end Reasoning Service (BRS) component for each scenario (averaged over all scenario runs)

Scenario	RSPS	LRS	BRS
baseline	131.94	0	0
1	143.86	2.44	0
2	179.68	7.56	1

a node on Virtual Wall 1 of the imec iLab.t testbeds Virtual Wall [64]. This node has a 2000 MHz dual-core AMD Opteron 2212 CPU and 4 GB DDR2 333 MHz RAM.

2.7 Evaluation results

For each evaluation scenario, Table 2.4 shows the average amount of incoming RDF events on each component. These results are now described in detail for each scenario.

In the baseline scenario, on average 131.94 events have come in on the RSPS. Except for three runs, 132 events have arrived over the 60 s: 60 light intensity observations, 60 sound observations, and 12 BLE observations (1 every 5 s). No threshold is crossed, so no event is sent to the LRS. Hence, the LRS and BRS have no incoming events.

In scenario 1, there have been on average 143.86 incoming events on the RSPS. During a period of 10 s, the light intensity threshold is crossed, and the `FilterLightIntensity` query, executing every 5 s on a window with a size of 6 s, generates either two or three symptoms. These symptoms are sent to the LRS. There, only the first incoming event triggers the creation of a `WarnNurseAction`. This is handled locally, so no event is sent to the BRS.

In scenario 2, on average 179.68 events have arrived at the RSPS. Now, the light intensity threshold is crossed during a period of 35 s. Hence, seven or eight symptoms are generated by the `FilterLightIntensity` query. The first incoming event at the LRS generates a `CallNurseAction`, which is sent to the BRS. Hence, only one event arrives at the BRS. By the time the assigned nurse arrives at the room, the final one or two outgoing events at the RSPS also contain a nurse BLE observation, next to the symptom. Hence, at the LRS, a `WarnNurseAction` is constructed in addition. Again, by definition, this is handled locally, so no additional event is sent to the BRS.

Each observation that leads to the construction of an action on the LRS needs to be handled by the system. In scenario 1, one `WarnNurseAction` is generated, which is handled locally. In scenario 2, first a `CallNurseAction` is sent to the BRS (scenario 2a), and then a local `WarnNurseAction` is generated (scenario 2b). In Figure 2.7, a boxplot of the total system latency is shown for these three situations.

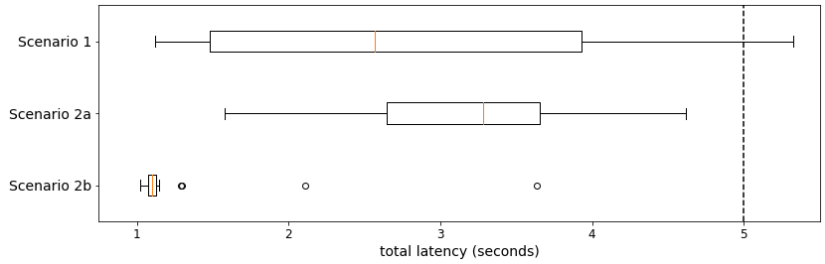


Figure 2.7: Boxplot showing the distribution, over all scenario runs, of the total system latency of three types of observations: the observation in scenario 1 causing a `WarnNurseAction` handled locally, the observation in scenario 2a leading to a `CallNurseAction` handled by the back-end, and another observation in scenario 2b that causes the creation of another `WarnNurseAction`. The vertical dashed line indicates the 5 s threshold, which is the targeted maximum system latency.

Figure 2.8 shows the average total system latency for the three cases, split into the different component and network latencies.

For scenario 1, i.e., the situation with a `WarnNurseAction`, the total system latency is below the targeted maximum latency of 5000 ms in more than 90% of the runs. In concrete, it only rises up to 325 ms above 5000 ms in three runs. As observed in Figure 2.8, the average system latency is 2703 ms. The two largest parts of this system latency are the RSPS latency and the RSPS-LRS network latency. Fluctuations in these two latencies cause the large spread in system latency. The distribution of the RSPS latency has a positive skew, i.e., its right tail is longer. There is a significant amount of outliers on this side, i.e., runs with a higher RSPS latency. This is true for all three scenarios. The LRS processing time is on average 98 ms, which is slightly larger than the average LRS latency of 69 ms.

In scenario 2a, a `CallNurseAction` is handled by the BRS. The observation needs to pass through all the main architecture components, causing the largest average system latency of 3142 ms. Again, the RSPS latency and RSPS-LRS network latency make up the largest part. The BRS latency is on average 384 ms. However, the average BRS processing time is 28413 ms, indicating an additional processing time after the query executions of approximately 28 s on average. By definition, this large additional processing time does not cause any additional latency.

Scenario 2b involves an additional `WarnNurseAction`. Here, the average system latency is only 1215 ms. The average RSPS-LRS network latency is significantly smaller compared to the other two cases. The average RSPS latency is slightly smaller, and the other average latency values are comparable.

Finally, note that the queuing time on the LRS and BRS components is negligible for all runs of all scenarios.

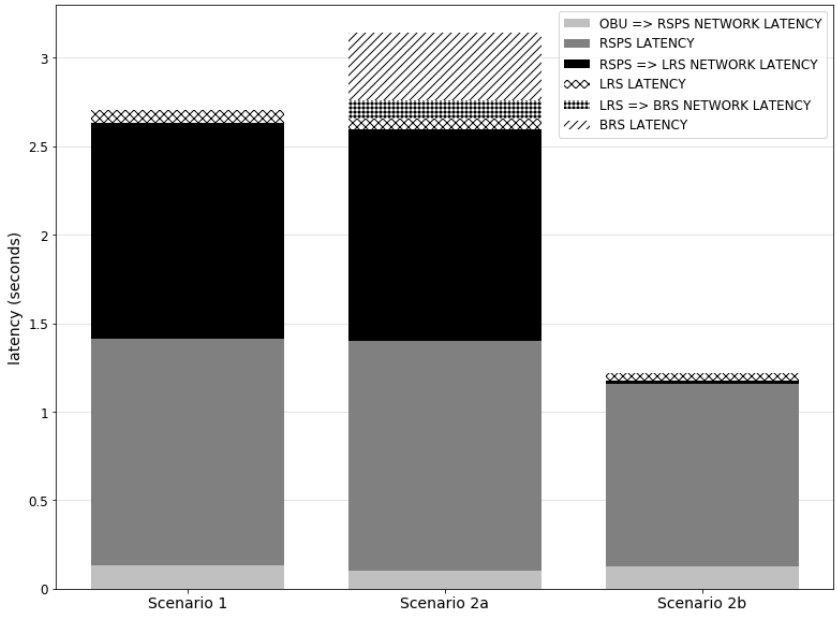


Figure 2.8: Bar plot showing the average total system latency of three types of observations, over all scenario runs: the observation in scenario 1 causing a `WarnNurseAction` handled locally, the observation in scenario 2a leading to a `CallNurseAction` handled by the back-end, and another observation in scenario 2b that causes the creation of another `WarnNurseAction`. For each situation, the different network and component latencies that sum up to the total system latency are indicated by the stacked bars.

2.8 Discussion

The results of the performance evaluation described in Section 2.7 allow to verify the evaluation goals explained in Section 2.6.1. As explained in the second evaluation goal, each nurse call assignment should be completed within 5 s after the fault occurrence, i.e., after the observation that crossed the threshold. Translating this to the evaluation results, this means that the system latency should be lower than 5 s. As can be observed in Figure 2.7, this was the case in all runs of scenario 2a for the `CallNurseAction`. However, one remark should be made concerning the simple `SelectNurse` query running on the BRS, instead of a complex oNCS. In fact, the time of a complex nurse call assignment algorithm should be added to the BRS latency. Considering such a very complex algorithm, such an assignment takes around 550 ms [65]. Adding this number, only three runs exceed the threshold of 5000 ms, and not by more than 250 ms. Ideally, the decision to warn a nurse is also taken within 5 s. Except for three runs in scenario 1, this requirement is also met.

A remark should however be made regarding the RSPS latency. This latency is directly dependent on the chosen query sliding step, i.e., the period between two executions of the query. This period defines the worst case scenario for the RSPS latency. In the case of the `FilterLightIntensity` query, which is similar to the `FilterSound` query in Listing 2.2, this sliding step is 5 s. The RSPS latency, equal to the RSPS processing time, is the time between the observation arrival and the start of the query execution, summed up with the duration of the query execution itself. If a threshold crossing observation arrives at the RSPS right after the start of the query execution, the first component of this sum can already be up to 5 s. Hence, to have more guarantees that the total system latency is always below 5 s, the frequency of the query execution should be increased. However, for this PoC, the `FilterLightIntensity` query creates a symptom for the most recent light intensity observation above the threshold. It is this observation that propagates through the system, and the associated observation time is used to calculate the system latency. When a threshold in light intensity is crossed, this will often be the case for multiple consecutive seconds, as is also the case in the evaluation scenarios. Hence, in most cases, when a threshold is crossed, this will be the case for the most recent observation in the window. As light intensity observations arrive at the RSPS every second, most RSPS processing times are around one second. In some cases, the most recent threshold crossing observation had arrived more than one second before the next query execution, explaining the RSPS processing times that are higher. In summary, the system latency is always below 5 s in the performed evaluation, but a few older observations in the window could already have been crossing the threshold.

Inspecting the breakdown of the total latency into component and network latencies in Figure 2.8, and associated numbers presented in Section 2.7 (evaluation goal 3), some other remarks can be made. First, the RSPS-LRS network latency is on

average quite high in the case of the `WarnNurseAction` in scenario 1 and the `CallNurseAction` in scenario 2a. These actions always correspond to the first outgoing event of the scenario at the RSPS. As components were restarted after each scenario run in the evaluation, this was always the first outgoing event at runtime. As can be observed in the case of the `WarnNurseAction` in scenario 2b, the RSPS-LRS system latency is way lower on average for the following outgoing events. In real-life scenarios, components will not be restarted often. Hence, this network latency will be smaller on average. Second, the additional processing time after the query executions on the BRS is 28 s. This is caused by the removal of the data event and consecutive incremental reasoning step in RDFox. This large processing time does not affect the BRS latency, but is nevertheless not ideal. Hence, further research should be done on the reasoning service to decrease this value. Third, queuing time on the BRS is negligible in the current evaluation set-up. In real scenarios, other components may also send events to the BRS, causing a longer average and worst case waiting time. As the amount of events arriving at the BRS is, however, small in the average situation, the real impact may be limited. Fourth, as Table 2.4 indicates, the amount of incoming events on the components is not equal in each scenario run. In the three scenarios, 132, 144 and 180 events are generated by the OBU, respectively. Depending on the exact timing of the query executions, the last threshold crossing observation is present in either one or two query execution windows. This explains why, for scenario 1, the amount of incoming events of the LRS varies between 2 and 3. Similarly for scenario 2, the LRS receives either seven or eight incoming events. In each run, the amount of incoming events on the BRS is however the same.

The execution of the evaluation scenarios shows that a cascading reasoning system is well-suited for the described pervasive health use case. When no alarming situation occurs (baseline scenario), all events are filtered out by the RSPS. In this case, it is useless to contact the LRS or BRS. In the case where a potential alarming situation occurs, the RSPS just sends the event to the LRS, as the RSPS performs no reasoning. The LRS will process the event and reason on the data to infer whether or not it is a real alarming situation, i.e., a fault. If a fault is inferred and can be handled locally, it will be handled locally, and the BRS will also not be contacted (scenario 1). In this use case, this is when a nurse is present in the room. In that case, the local component knows perfectly what to do without any additional missing information: it should warn the nurse of the fault. The BRS is contacted (scenario 2) only if no nurse is present, as the LRS does not know by itself what nurses are available, where they currently are, what their current occupation is, etc. The BRS does know this information and is best-suited to decide on which nurse to call. In real-life situations, the BRS may require additional information to select and assign a nurse to a call. For example, every BLE observation of any nurse can be sent via the RSPS and LRS to the BRS, using additional simple queries. In this way, the BRS knows for each nurse when he/she is in a hospital room.

In its current implementation, the system performs two continuous checks for a concussion patient: it observes the light intensity and sound in the room. In real-life situations, the system may also need to monitor other alarming situations, e.g., when a patient has fallen on the ground. Adding these additional checks is easily possible by incorporating the required sensors, extending the domain ontology accordingly, providing the appropriate context information, and registering the corresponding queries on the different components. In some situations, priorities concerning the alarming situations and corresponding actions may need to be defined. For example, if a more severe alarming situation occurs, this may overrule a previous nurse call by sending a new nurse call with higher priority. Because the LRS and BRS components allow for defining a priority ordered list of event-based processing queries, such flexibility can be incorporated into the system.

The presented system has a few drawbacks and limitations. In the current architecture, no loss of connectivity to the cloud is assumed. To solve this, a buffer component may be incorporated between every LRS-BRS connection to cope with connectivity losses. This component may run on the same node as the LRS component, or may even be incorporated in the LRS by updating the reasoning service presented in Section 2.5.3.1. In case of connectivity losses, outgoing events will be buffered, and will be sent in order of arrival to the observing BRS components as soon as the connection is reestablished. Moreover, the current system is not capable of detecting false alarms. For example, severe noise may be present in the sensor observations, causing a series of light intensity observations to be quite stable with a random outlier above the threshold. In the current system, a symptom and potential fault will be generated. Future research should investigate how the system and the queries can be made more intelligent to be able to detect outliers and false alarms. Also when light intensity or sound values are gradually increasing, they can be fluctuating around the threshold at some point. Similarly, the system should be made more intelligent to avoid generating random symptoms and alarms depending on the observed noisy sensor value.

The presented evaluation has been performed in ideal simulation conditions: there was only small Gaussian noise added to the sensor observations, and only one patient and OBU were considered. As explained in the previous paragraph, more severe and unpredictable noise may cause issues that need to be solved in future research. The impact of simulating with only one patient in the used architectural set-up is small locally, as a consequence of combining Fog computing with cascading reasoning: an RSPS is only responsible for one patient, and an LRS for all patients in a single hospital room. The impact on the BRS will be larger, but investigating the scaling of a cloud component with an oNCS has already been investigated in previous research [62].

The concept of cascading reasoning and Fog computing has a big advantage with respect to the amount of events sent to the different components. For the evaluation

scenarios, this is clearly shown in Table 2.4. These scenarios highlight specific situations. However, it is explicitly interesting to investigate a real-life situation. Assuming the evaluation set-up of Section 2.6 with a single-person room, 7920 observation events are generated per hour by each OBU: 3600 light intensity observations, 3600 sound observations and 720 BLE tag observations, assuming that only the patient is present in the room. Assuming a small hospital with 20 single-person rooms, 158,400 events are generated per hour by all OBUs. In a real-life hospital setting, multiple other sensors will also be producing observations, making the actual value of events per hour a multiple of this value. In another set-up with only one BRS component performing all reasoning and processing, all of these events will arrive at this BRS. This puts a high burden on the BRS. Such a set-up is comparable to many of the recent cloud-based AAL solutions addressed in Section 2.2.3. In the cascading set-up, the amount of events arriving at the BRS will be a few orders of magnitude smaller, ranging from a few tens to a few hundreds per hour. At the LRS, the amount of events will be another order of magnitude larger, but still way smaller than the original amount of events received by the RSPS. This cascading set-up has many advantages. The events do not need to be processed any longer by a single component, avoiding a single point of failure. Events that can be processed locally will be processed locally, improving the autonomy of the system components. Moreover, the network traffic is reduced, decreasing the transmission cost and increasing the available bandwidth. The back-end and network resources are saved for situations with higher urgency and priority. This all improves the overall responsiveness, throughput and Quality-of-Service of the system. Note that some of the current AAL solutions presented in Section 2.2.3 also partly have some of these advantages. However, recall that none of these solutions combines stream and cascading reasoning. As a consequence, they are not able to perform real-time analysis on the data streams to make time-critical decisions, which is one of the main objectives of the presented system.

The advantages mentioned in the preceding paragraph can all be relevant for several use cases, inside and outside healthcare. Referring to the smart healthcare requirements addressed in Section 2.1.1, the presented cascading reasoning platform offers a solution to them. Personalized decision-making is possible in a time-critical way, as shown by the performance evaluation results. Given the example of the presented PoC use case, alarming situations for a patient can be detected locally. While this detection is processed by the BRS to call a nurse to the room, local and edge components can already take automatic action to partially solve the alarming situation, e.g., by dimming the lights for a concussion patient. Moreover, the architecture can cope with a limited amount of resources. Less expensive hardware should be invested in for local and edge components, while investment in more expensive high-level devices for the BRS components can be limited. Furthermore, the generic architecture has a high degree of configurability, allowing for flexible privacy management. For example, if required, sensitive data can already be processed locally, so that it does not need to be

transmitted over the backbone network to the back-end. By fulfilling these requirements and offering a solution to many of the issues that exist in ambient-intelligent healthcare, the presented cascading reasoning platform and architecture have the potential to be incorporated in real-life healthcare settings.

2.9 Conclusions and future work

In this chapter, a cascading reasoning framework is proposed, which can be used to support responsive ambient-intelligent healthcare interventions. A generic cascading architecture is presented that allows for constructing a pipeline of reasoning components, which can be hosted locally, in the edge of the network, and in the cloud, corresponding to the Fog computing principles. The architecture is implemented and evaluated on a pervasive health use case situated in hospital care, where medically diagnosed patients are constantly monitored. A performance evaluation has shown that the total system latency is lower than 5 s in almost all cases, allowing for fast intervention by a nurse in case of an alarming situation. It is discussed how the cascading reasoning platform solves existing issues in smart healthcare, by offering the possibility to perform personalized time-critical decision-making, by enabling the usage of heterogeneous devices with limited resources, and by allowing for flexible privacy management. Additional advantages include reduced network traffic, saving of back-end resources for high priority situations, improved responsiveness and autonomy, and removal of a single point of failure. Future work offers some interesting pathways. First, it should be researched how to deal with connectivity losses and noisy sensor observations, which are current system drawbacks. Second, a large scale evaluation of the platform should be performed with multiple devices and different healthcare scenarios in realistic conditions. To this end, data collection of representative patient profiles and healthcare scenarios is currently ongoing. Third, the framework can be extended to adaptively distribute a federated cascading reasoning system across the IoT fog, also taking into account the scaling of the full system.

Funding

Femke Ongenae is funded by a BOF postdoc grant from Ghent University. Part of this research was funded by the FWO SBO grant number 150038 (DiSSeCt), and the imec ICON CONAMO. CONAMO is funded by imec, VLAIO, Rombit, Energy Lab and VRT.

Availability of data and materials

The designed continuous care ontology is publicly available at https://github.com/IBCNServices/cascading-reasoning-framework. The original ACCIO ontology, which is the starting point of the designed ontology, is publicly available at https://github.com/IBCNServices/Accio-Ontology/tree/gh-pages.

References

[1] J.-C. Burgelman and Y. Punie. *Information, society and technology*. In E. Aarts and J. Encarnação, editors, True Visions: The Emergence of Ambient Intelligence, pages 17–33. Springer, Berlin, Heidelberg, 2006. doi:10.1007/978-3-540-28974-6_2.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. *Internet of Things: A survey on enabling technologies, protocols, and applications*. IEEE Communications Surveys & Tutorials, 17(4):2347–2376, 2015. doi:10.1109/COMST.2015.2444095.

[3] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. *Context Aware Computing for The Internet of Things: A Survey*. IEEE Communications Surveys & Tutorials, 16(1):414–454, 2014. doi:10.1109/SURV.2013.042313.00197.

[4] F. Ongenae, J. Famaey, S. Verstichel, S. De Zutter, S. Latré, A. Ackaert, P. Verhoeve, and F. De Turck. *Ambient-aware continuous care through semantic context dissemination*. BMC Medical Informatics and Decision Making, 14(1), 2014. doi:10.1186/1472-6947-14-97.

[5] *Internet of Medical Things, Forecast to 2021*. Accessed: 2018-08-13. Available from: https://store.frost.com/internet-of-medical-things-forecast-to-2021.html.

[6] C. C. Aggarwal, N. Ashish, and A. Sheth. *The Internet of Things: A Survey from the Data-Centric Perspective*. In C. C. Aggarwal, editor, Managing and Mining Sensor Data, pages 383–428. Springer US, 2013. doi:10.1007/978-1-4614-6309-2_12.

[7] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. *Semantics for the Internet of Things: Early Progress and Back to the Future*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012. doi:10.4018/jswis.2012010101.

[8] T. R. Gruber. *A translation approach to portable ontology specifications*. Knowledge Acquisition, 5(2):199–220, 1993. doi:10.1006/knac.1993.1008.

[9] M. Compton, P. Barnaghi, L. Bermudez, R. García-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. Le Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor. *The SSN ontology of the W3C semantic sensor network incubator group*. Journal of Web Semantics, 17:25–32, 2012. doi:10.1016/j.websem.2012.05.003.

[10] SNOMED International. *SNOMED*. Accessed: 2018-10-18. Available from: https://www.snomed.org.

[11] HL7. *FHIR*. Accessed: 2018-10-18. Available from: http://hl7.org/fhir/index.html.

[12] C. Bizer, T. Heath, and T. Berners-Lee. *Linked data: The story so far*. In A. Sheth, editor, Semantic Services, Interoperability and Web Applications: Emerging Concepts, pages 205–227. IGI Global, 2011. doi:10.4018/978-1-60960-593-3.ch008.

[13] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang. *HermiT: an OWL 2 reasoner*. Journal of Automated Reasoning, 53(3):245–269, 2014. doi:10.1007/s10817-014-9305-1.

[14] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. *RDFox: A highly-scalable RDF store*. In The Semantic Web - ISWC 2015: Proceedings, Part II of the 14th International Semantic Web Conference, pages 3–20, Cham, Switzerland, 2015. Springer. doi:10.1007/978-3-319-25010-6_1.

[15] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. *Stream reasoning: A survey and outlook*. Data Science, 1(1-2):59–83, 2017. doi:10.3233/DS-170006.

[16] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. *OWL 2 Web Ontology Language Profiles*. W3C Editor's Draft, World Wide Web Consortium (W3C), 2009. Available from: https://www.w3.org/2007/OWL/draft/ED-owl2-profiles-20090610/.

[17] M. A. Sahi, H. Abbas, K. Saleem, X. Yang, A. Derhab, M. A. Orgun, W. Iqbal, I. Rashid, and A. Yaseen. *Privacy Preservation in e-Healthcare Environments: State of the Art and Future Directions*. IEEE Access, 6:464–478, 2018. doi:10.1109/ACCESS.2017.2767561.

[18] X. Su, P. Li, J. Riekki, X. Liu, J. Kiljander, J.-P. Soininen, C. Prehofer, H. Flores, and Y. Li. *Distribution of Semantic Reasoning on the Edge of Internet of Things*. In 2018 IEEE International Conference on Pervasive Computing and Communications (PerCom), 2018. doi:10.1109/PERCOM.2018.8444596.

[19] A. Margara, J. Urbani, F. Van Harmelen, and H. Bal. *Streaming the web: Reasoning over dynamic data*. Journal of Web Semantics, 25:24–44, 2014. doi:10.1016/j.websem.2014.02.001.

[20] X. Su, E. Gilman, P. Wetz, J. Riekki, Y. Zuo, and T. Leppänen. *Stream reasoning for the Internet of Things: Challenges and gap analysis*. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), pages 1–10, New York, NY, USA, 2016. Association for Computing Machinery (ACM). doi:10.1145/2912845.2912853.

[21] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. *C-SPARQL: a continuous query language for RDF data streams*. International Journal of Semantic Computing, 4(1):3–25, 2010. doi:10.1142/S1793351X10000936.

[22] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. *A native and adaptive approach for unified processing of linked streams and linked data*. In The Semantic Web - ISWC 2011: Proceedings, Part I of the 10th International Semantic Web Conference, pages 370–388, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-25073-6_24.

[23] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. *EP-SPARQL: a unified language for event processing and stream reasoning*. In Proceedings of the 20th International Conference on World Wide Web (WWW 2011), pages 635–644, New York, NY, USA, 2011. Association for Computing Machinery (ACM). doi:10.1145/1963405.1963495.

[24] J.-P. Calbimonte, O. Corcho, and A. J. Gray. *Enabling ontology-based access to streaming data sources*. In The Semantic Web – ISWC 2010, pages 96–111. Springer, 2010. doi:10.1007/978-3-642-17746-0_7.

[25] S. Komazec, D. Cerri, and D. Fensel. *Sparkwave: continuous schema-enhanced pattern matching over RDF data streams*. In Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS 2012), pages 58–68, New York, NY, USA, 2012. Association for Computing Machinery (ACM). doi:10.1145/2335484.2335491.

[26] M. Rinne, E. Nuutila, and S. Törmä. *INSTANS: high-performance event processing with standard RDF and SPARQL*. In ISWC-PD'12: Proceedings of the 2012th International Conference on Posters & Demonstrations Track – Volume 914, pages 101–104, 2012. Available from: https://dl.acm.org/doi/10.5555/2887379.2887405.

[27] C. L. Forgy. *Rete: A fast algorithm for the many pattern/ many object pattern match problem*. Artificial Intelligence, 19(1):17–37, 1982. doi:10.1016/0004-3702(82)90020-0.

[28] S. Germano, T.-L. Pham, and A. Mileo. *Web stream reasoning in practice: on the expressivity vs. scalability tradeoff*. In RR 2015: Web Reasoning and Rule Systems, pages 105–112. Springer, 2015. doi:10.1007/978-3-319-22002-4_9.

[29] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. Van Harmelen. *Towards expressive stream reasoning*. In Semantic Challenges in Sensor Networks, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. doi:10.4230/DagSemProc.10042.4.

[30] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya. *Chapter 4 – Fog computing: Principles, architectures, and applications*. In R. Buyya and A. Vahid Dastjerdi, editors, Internet of Things, pages 61–75. Morgan Kaufmann, 2016. doi:10.1016/B978-0-12-805395-9.00004-6.

[31] R. Mahmud, R. Kotagiri, and R. Buyya. *Fog Computing: A Taxonomy, Survey and Future Directions*. In B. Di Martino, K.-C. Li, L. T. Yang, and A. Esposito, editors, Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives, pages 103–130. Springer Singapore, 2018. doi:10.1007/978-981-10-5861-5_5.

[32] J. Gedeon, J. Heuschkel, L. Wang, and M. Mühlhäuser. *Fog Computing: Current Research and Future Challenges*. In KuVS-Fachgespräch Fog Computing 2018, 2018.

[33] O. Skarlat, K. Bachmann, and S. Schulte. *FogFrame: IoT Service Deployment and Execution in the Fog*, 2018. Lecture notes at KuVS-Fachgespräch Fog Computing 2018. Available from: https://dsg.tuwien.ac.at/docs/03_Skarlat_FogFrame.pdf.

[34] A. Gyrard, S. K. Datta, C. Bonnet, and K. Boudaoud. *A semantic engine for Internet of Things: cloud, mobile devices and gateways*. In 2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, pages 336–341. IEEE, 2015. doi:10.1109/IMIS.2015.83.

[35] Y. A. Sedira, R. Tommasini, and E. Della Valle. *MobileWave: Publishing RDF Streams From SmartPhones*. In Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks, co-located with 16th International Semantic Web Conference (ISWC 2017). Springer, 2017. Available from: https://ceur-ws.org/Vol-1963/paper536.pdf.

[36] W. Van Woensel and S. S. R. Abidi. *Optimizing Semantic Reasoning on Memory-Constrained Platforms Using the RETE Algorithm*. In ESWC 2018: The Semantic Web, pages 682–696. Springer, 2018. doi:10.1007/978-3-319-93417-4_44.

[37] V. Charpenay, S. Käbisch, and H. Kosch. *Towards a Binary Object Notation for RDF*. In ESWC 2018: The Semantic Web, pages 97–111. Springer, 2018. doi:10.1007/978-3-319-93417-4_7.

[38] N. M. Garcia and J. J. P. C. Rodrigues, editors. *Ambient Assisted Living*. CRC Press, 2015. doi:10.1201/b18520.

[39] R. I. Goleva, I. Ganchev, C. Dobre, N. Garcia, and C. Valderrama, editors. *Enhanced Living Environments: From models to technologies*. Institution of Engineering and Technology, 2017. doi:10.1049/PBHE010E.

[40] R. I. Goleva, N. M. Garcia, C. X. Mavromoustakis, C. Dobre, G. Mastorakis, R. Stainov, I. Chorbev, and V. Trajkovik. *Chapter 8 – AAL and ELE Platform Architecture*. In C. Dobre, C. Mavromoustakis, N. Garcia, R. Goleva, and G. Mastorakis, editors, Ambient Assisted Living and Enhanced Living Environments: Principles, Technologies and Control, pages 171–209. Butterworth-Heinemann, 2017. doi:10.1016/B978-0-12-805195-5.00008-9.

[41] A. Moawad. *Towards Ambient Intelligent Applications Using Models@run.time And Machine Learning For Context-Awareness*. PhD thesis, University of Luxembourg, 2016. Available from: http://hdl.handle.net/10993/23746.

[42] X. B. Valencia, D. B. Torres, C. P. Rodriguez, D. H. Peluffo-Ordóñez, M. A. Becerra, and A. E. Castro-Ospina. *Case-Based Reasoning Systems for Medical Applications with Improved Adaptation and Recovery Stages*. In IWBBIO 2018: Bioinformatics and Biomedical Engineering, pages 26–38. Springer, 2018. doi:10.1007/978-3-319-78723-7_3.

[43] C. M. Nguyen, R. Sebastiani, P. Giorgini, and J. Mylopoulos. *Multi-objective reasoning with constrained goal models*. Requirements Engineering, 23:189–225, 2018. doi:10.1007/s00766-016-0263-5.

[44] S. Jabbar, F. Ullah, S. Khalid, M. Khan, and K. Han. *Semantic interoperability in heterogeneous IoT infrastructure for healthcare*. Wireless Communications and Mobile Computing, 2017, 2017. doi:10.1155/2017/9731806.

[45] H. Mohammadhassanzadeh, S. R. Abidi, M. S. Shah, M. Karamollahi, and S. S. R. Abidi. *SeDAn: a Plausible Reasoning Approach for Semantics-based Data Analytics in Healthcare*. In Proceedings of the Workshop on Artificial Intelligence with Application in Health, co-located with the 16th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2017), 2017. Available from: https://ceur-ws.org/Vol-1982/paper7.pdf.

[46] F. De Backere, P. Bonte, S. Verstichel, F. Ongenae, and F. De Turck. *The OCare-Platform: A context-aware system to support independent living*. Computer Methods and Programs in Biomedicine, 140:111–120, 2017. doi:10.1016/j.cmpb.2016.11.008.

[47] N. Lasierra, A. Alesanco, and J. Garcia. *Designing an architecture for monitoring patients at home: ontologies and web services for clinical and technical management integration*. IEEE Journal of Biomedical and Health Informatics, 18(3):896–906, 2014. doi:10.1109/JBHI.2013.2283268.

[48] H. Kuijs, C. Rosencrantz, and C. Reich. *A context-aware, intelligent and flexible ambient assisted living platform architecture*. Cloud Computing, 2015.

[49] A. Forkan, I. Khalil, and Z. Tari. *CoCaMAAL: A cloud-oriented context-aware middleware in ambient assisted living*. Future Generation Computer Systems, 35:114–127, 2014. doi:10.1016/j.future.2013.07.009.

[50] F. Paganelli and D. Giuli. *An ontology-based system for context-aware and configurable services to support home-based continuous care*. IEEE Transactions on Information Technology in Biomedicine, 15(2):324–333, 2011. doi:10.1109/TITB.2010.2091649.

[51] M. Amoretti, S. Copelli, F. Wientapper, F. Furfari, S. Lenzi, and S. Chessa. *Sensor data fusion for activity monitoring in the PERSONA ambient assisted living project*. Journal of Ambient Intelligence and Humanized Computing, 4:67–84, 2013. doi:10.1007/s12652-011-0095-6.

[52] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. *Linking Data to Ontologies*. In Journal on Data Semantics X, pages 133–173, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-77688-8_5.

[53] *Designed Continuous Care Ontology*, 2018. Accessed: 2018-10-18. Available from: https://github.com/IBCNServices/cascading-reasoning-framework.

[54] *ACCIO Ontology*, 2018. Accessed: 2018-10-18. Available from: https://github.com/IBCNServices/Accio-Ontology/tree/gh-pages.

[55] F. Ongenae, P. Duysburgh, N. Sulmon, M. Verstraete, L. Bleumers, S. De Zutter, S. Verstichel, A. Ackaert, A. Jacobs, and F. De Turck. *An ontology co-design method for the co-creation of a continuous care ontology*. Applied Ontology, 9(1):27–64, 2014. doi:10.3233/AO-140131.

[56] L. Daniele, F. den Hartog, and J. Roes. *Created in Close Interaction with the Industry: The Smart Appliances REFerence (SAREF) Ontology*. In Formal Ontologies Meet Industry, pages 100–112, Cham, Switzerland, 2015. Springer. doi:10.1007/978-3-319-21545-7_9.

[57] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. *Ontop: Answering SPARQL queries over relational databases*. Semantic Web, 8(3):471–487, 2017. doi:10.3233/SW-160217.

[58] *Stardog*. Accessed: 2018-10-18. Available from: https://www.stardog.com.

[59] *RSP Service Interface for C-SPARQL*. Accessed: 2018-10-18. Available from: https://github.com/streamreasoning/rsp-services-csparql/.

[60] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. *OWL 2 Web Ontology Language Profiles (Second Edition) – OWL 2 RL Profile*. W3C Recommendation, World Wide Web Consortium (W3C), 2012. Available from: https://www.w3.org/TR/owl2-profiles/#OWL_2_RL.

[61] J. Nelis, T. Verschueren, D. Verslype, and C. Develder. *Dyamand: dynamic, adaptive management of networks and devices*. In 37th Annual IEEE Conference on Local Computer Networks, pages 192–195. IEEE, 2012. doi:10.1109/LCN.2012.6423604.

[62] F. Ongenae, D. Myny, T. Dhaene, T. Defloor, D. Van Goubergen, P. Verhoeve, J. Decruyenaere, and F. De Turck. *An ontology-based nurse call management system (oNCS) with probabilistic priority assessment*. BMC Health Services Research, 11(1), 2011. doi:10.1186/1472-6963-11-26.

[63] Intel Corporation. *Specifications of an Intel NUC, Model D54250WYKH*. Accessed: 2018-10-18. Available from: https://ark.intel.com/products/81164/Intel-NUC-Kit-D54250WYKH.

[64] Ghent University – imec. *iLab.t Virtual Wall*, 2018. Accessed: 2018-10-18. Available from: https://doc.ilabt.imec.be/ilabt/.

[65] P. Bonte, F. Ongenae, J. Schaballie, B. De Meester, D. Arndt, W. Dereuddre, J. Bhatti, S. Verstichel, R. Verborgh, R. Van de Walle, E. Mannens, and F. De Turck. *Evaluation and optimized usage of OWL 2 reasoners in an event-based eHealth context*. In Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation (ORE-2015), co-located with the 28th International Workshop on Description Logics (DL 2015), pages 1–7, 2015. Available from: https://ceur-ws.org/Vol-1387/paper_6.pdf.

3

Context-Aware Query Derivation for IoT Data Streams with DIVIDE Enabling Privacy By Design

In Chapter 2, a generic cascading reasoning framework was presented. This framework distributes stream reasoning queries across the different devices in an IoT network, involving both local, edge and central devices. The distribution of queries in this framework is however still static: queries need to be configured by the end users, and a reconfiguration is required if the use case context changes. Therefore, this chapter proposes a solution that integrates adaptiveness into the cascading reasoning framework. This solution is DIVIDE, which is a component that can be integrated in a semantic IoT platform that applies the cascading reasoning framework. The chapter presents the methodological design of DIVIDE, which automatically ensures that efficient, context-aware queries are deployed on the platform's stream processing components at all times, by deriving the conditions and window parameters of the contextually relevant queries every time the use case context changes. Moreover, the chapter explains how DIVIDE enables privacy by design. The chapter also presents a first implementation of DIVIDE, and evaluates it on the homecare monitoring use case UC2, with a focus on the in-home activity monitoring of patients. At the end of this chapter, four addenda are added that contain additional information related to this chapter.

This chapter addresses research challenge RCH2 ("Adaptive configuration of stream processing queries based on use case context, enabling privacy by design") by discussing research contribution RCO2. Moreover, it also addresses research challenge RCH1 ("Performant & responsive

real-time stream reasoning with local autonomy across a heterogeneous IoT network") by building further on research contribution RC01. The chapter validates research hypothesis RH3: "The methodological design of a semantic IoT platform component that derives and configures the conditions & window parameters of stream processing queries whenever the use case context changes will result in adaptive, context-aware queries that only require simple filtering and thus enable the local filtering of contextually relevant events in less than 5 seconds on low-end IoT devices with few resources. This will fully remove the required manual query reconfiguration effort when changes to the use case context occur.". Moreover, the chapter also validates research hypothesis RH4: "The methodological design of a semantic IoT platform component that enables privacy by design will let the end user in 100% control about which data abstractions can be sent over the network and which data is not leaving the local environments of the IoT network, while maintaining an overhead to adapt the queries based on changing use case context that is at most 1 order of magnitude (i.e., 10 times) higher than the execution time of semantic queries on equivalent state-of-the-art real-time reasoning set-ups.".

⋆ ⋆ ⋆

M. De Brouwer, B. Steenwinckel, Z. Fang, M. Stojchevska, P. Bonte, F. De Turck, S. Van Hoecke, and F. Ongenae

Published in Semantic Web Journal, Volume 14, Issue 5, May 2023.

Abstract

Integrating Internet of Things (IoT) sensor data from heterogeneous sources with domain knowledge and context information in real-time is a challenging task in IoT healthcare data management applications that can be solved with semantics. Existing IoT platforms often have issues with preserving the privacy of patient data. Moreover, configuring and managing context-aware stream processing queries in semantic IoT platforms requires much manual, labor-intensive effort. Generic queries can deal with context changes but often lead to performance issues caused by the need for expressive real-time semantic reasoning. In addition, query window parameters are part of the manual configuration and cannot be made context-dependent. To tackle these problems, this chapter presents DIVIDE, a component for a semantic IoT platform that adaptively derives and manages the queries of the platform's stream processing components in a context-aware and scalable manner, and that enables privacy by design. By performing semantic reasoning to derive the queries when context changes are observed, their real-time evaluation does require any reasoning. The results of an evaluation on a homecare monitoring use case demonstrate how activity detection queries derived with DIVIDE can be evaluated in on average less than 3.7 seconds and can therefore successfully run on low-end IoT devices.

3.1 Introduction

3.1.1 Background

In the healthcare domain, many applications involve a large collection of Internet of Things (IoT) devices and sensors [1]. Many of those systems typically focus on the real-time monitoring of patients in hospitals, nursing homes, homecare or elsewhere. In such systems, patients and their environment are being equipped with different devices and sensors for following up on the patients' conditions, diseases and treatments in a personalized, context-aware way. This is achieved by integrating the data collected by the IoT devices with existing domain knowledge and context information. As such, analyzing this combination of data sources jointly allows a system to extract meaningful insights and actuate on them [2].

Integrating and analyzing the IoT data with domain knowledge and context information in a real-time context is a challenging task. This is due to the typically high volume, variety and velocity of the different data sources [3]. To deal with these challenges, semantic IoT platforms can be deployed [4]. They generally contain stream processing components that integrate and analyze the different data sources by continuously evaluating semantic queries. To deploy this, Semantic Web technologies are typically employed: ontologies are designed to integrate and model the data from different heterogeneous sources and its relationships and properties in a common, machine-interpretable format, and existing stream reasoning techniques are used by the data stream processing components [5].

Currently, the configuration and management of queries that run on the stream processing components of a semantic IoT platform are manual tasks that require a lot of effort from the end user. In the typical IoT applications in healthcare, those queries should be context-aware: the context information determines which sensors and devices should be monitored by the query, for example to filter specific events to send to other components for further analysis. For example, a patient's diagnosis in the Electronic Health Record (EHR) determines the monitoring tasks that should be performed in the patient's hospital room, while the indoor location (room) of the patient in a homecare monitoring environment determines which in-home activities can be monitored. Changes in this context information regularly occur. For example, the profile information of patients in their EHR can be updated, or the in-home location of the patient can evolve over time. Hence, the management of the queries should be able to deal with such context changes. Currently, no semantic IoT platform component exists that allows to configure, derive and manage the platform's queries in an automated, adaptive way. Therefore, platforms typically apply one of two existing approaches to achieve this.

The first approach to introduce context-awareness into semantic queries is by defining them in a generic fashion. A generic query uses generic ontology concepts

in its definitions to perform multiple contextually relevant tasks. This way, semantic reasoners will reason in real-time on all available streaming, context and domain knowledge data to determine the contextually relevant sensors and devices to which the query is applicable. The advantage of this approach is that such queries are prepared to deal with contextual changes: due to their generic nature, they should not be updated often. However, the disadvantage of highly generic queries is the high computational complexity of the semantic reasoning during their evaluation. This is caused by complex ontologies in IoT domains such as healthcare that require expressive reasoning [6]. In healthcare applications that involve a large number of sensors, it is practically challenging to do this in real-time: queries take longer to evaluate, causing lower performance and difficulty to keep up with the required query execution frequencies. Typically, central components in an IoT platform have more resources and are therefore more likely to overcome this challenge. However, running all queries on central components would require all generated IoT streaming data to be sent over the network, causing the network to be highly congested all the time. In addition, the central server resources would be constantly in use, and local decision making would no longer be possible. Importantly, this would also imply no flexibility in preserving the patient's privacy by keeping sensitive data locally. Looking at local and edge IoT devices to run those generic queries instead, resources are typically lower, making the performance challenges an even bigger issue of the generic query approach.

An alternative approach that can be adopted is installing multiple specific queries on the stream processing components that filter the contextually relevant sensors for one specific task. Evaluating such non-generic queries reduces the required semantic reasoning effort, solving the performance issues of the generic approach. However, this approach even further increases the required manual query configuration and management effort for the end user: whenever the context changes, the queries should be manually updated. This is infeasible to do in practice. As a consequence, current platforms do not apply this approach often and mostly work with generic queries instead.

Moreover, the definition of generic stream processing queries does not contain any means to make the window parameters of the query dependent on the application context and domain knowledge. Currently, an end user should configure these query parameters, and cannot let the system define them based on the data. This can be a problem in some specific monitoring cases. For example, the size of the data window on which a monitoring task such as in-home activity detection should be executed, may depend on the type of task, and therefore be defined in the domain knowledge. Another example is when the execution frequency of a monitoring task depends on certain contextual events happening in the patient's environment.

In addition, preserving the privacy of the patients is of utmost importance in healthcare systems [7]. In IoT platforms, lots of the data generated by the IoT devices can contain privacy-sensitive information. Depending on where the data processing components are being hosted, this privacy-sensitive data may have to be sent over the IoT network, potentially exposing it to the outside world. Therefore, the IoT data is ideally processed close to where it is generated to reduce the amount of information sent over the network as much as possible. With regards to this, a semantic IoT platform should enable privacy by design [8]: it should allow an end user to build privacy by design into an application by precisely controlling which data is kept locally, and which data is sent over the network.

Finally, a semantic IoT platform component that would solve the aforementioned issues, should also be practically usable. Currently, existing semantic IoT healthcare platforms use semantic reasoners or stream reasoners that are configured with existing sets of generic semantic queries [2]. Defining such queries and ensuring their correctness is a delicate and time-consuming task. Hence, a new component should not introduce a completely different means of defining generic queries, but instead reduce the required changes to these definitions to a minimum. This implies that it should start from the generic definition of stream processing queries. Moreover, the other configuration tasks of the component should also be as minimal as possible to increase overall usability.

3.1.2 Research objectives and contribution

In summary, there is a need for a semantic IoT platform component that fulfills the different requirements tackled in the previous subsection, so that it can be applied in a healthcare data management system. Hence, we set the following research objectives for the design of such an additional semantic IoT platform component:

1. The component should reduce the manual, labor-intensive query configuration effort by managing the queries on the platform's stream processing components in an automated, adaptive and context-aware way.

2. The evaluation of queries managed by the component should be performant, also on low-end IoT edge devices with fewer resources. Network congestion and overuse of central resources should be avoided.

3. The component should allow for the query window parameters to be context-dependent.

4. The component should enable privacy by design: it should allow end users to integrate privacy by design into an application by defining, on different levels of abstraction, which data is kept locally and which parts of the data can be sent over the network.

5. The component should be practically usable, minimizing the effort to integrate it into an existing system.

This chapter presents DIVIDE, a semantic IoT platform component that we have designed to achieve the presented research objectives. DIVIDE automatically and adaptively derives and manages the contextually relevant specific queries for the platform's stream processing components, by performing semantic reasoning with a generic query definition whenever contextual changes occur. As a result, the derived queries will efficiently monitor the relevant IoT sensors and devices in real-time, and still do not require any real-time reasoning during their evaluation.

The contribution of this chapter is the methodological design and proof-of-concept of the DIVIDE component, fulfilling the requirements associated with the above research objectives. In the chapter, DIVIDE is applied and evaluated on a realistic homecare monitoring use case, to demonstrate how it can be used in a practical IoT application context that works with privacy-sensitive information.

3.1.3 Chapter organization

The remainder of this chapter is structured as follows. Section 3.2 discusses some related work. In Section 3.3, the eHealth use case scenario is further explained, translated into the technical system set-up, and semantically described with an ontology. Section 3.4 presents a general overview of the DIVIDE system. Further functional and algorithmic details of DIVIDE are provided in Section 3.5 and Section 3.6 using the running use case example, while Section 3.7 zooms in on the technical implementation of DIVIDE. Section 3.8 describes the evaluation set-up with the different evaluation scenarios and hardware set-up. Results of the evaluations are presented in Section 3.9, and further discussed in Section 3.10. Finally, Section 3.11 concludes the main findings of the chapter and highlights future work.

3.2 Related work

3.2.1 Semantic Web, stream processing and stream reasoning

Using Semantic Web technologies such as the Resource Description Framework (RDF) and the Web Ontology Language (OWL), heterogeneous data sources can be consolidated and semantically enriched into a machine-interpretable representation using ontologies [4]. An ontology is a model that semantically describes all domain-specific knowledge by defining domain concepts and their relations and attributes. Within RDF, an Internationalized Resource Identifier (IRI) is used to refer to every resource defined in an ontology [9]. Semantic reasoners can interpret semantic data to derive new knowledge based on the definitions in the ontologies. The complexity of the semantic reasoning depends on the expressivity of the underlying ontology [10].

Different ontology languages exist. They range from RDFS, which has the lowest expressivity, to OWL 2 DL, which has the highest expressivity.

RDFox [11] and VLog [12] are state-of-the-art OWL 2 RL reasoners. OWL 2 RL contains all constructs that can be evaluated by a rule engine. These constructs can be expressed by simple Datalog rules. By design, these engines are not able to handle streaming data. However, RDFox can also run on a Raspberry Pi, and any ARM-based IoT edge device in general. In addition, previous research has shown it can also successfully run on a smartphone [13].

Notation3 Logic (N3) [14] is a rule-based logic that is often used to write down RDF. N3 is a superset of RDF/Turtle [15], which implies that any valid RDF/Turtle definitions are valid N3 as well.

Stream Reasoning (SR) [5] state-of-the-art contains three main approaches: Continuous Processing (CP) engines, Reasoning Over Time (ROT) frameworks and Reasoning About Time (RAT) frameworks. CP engines have continuous semantics, high throughput, and low latency but do not perform reasoning. ROT frameworks solve reasoning tasks continuously with high throughput and low latency, but do not consider time. RAT frameworks do consider time in the reasoning task, but may lack reactivity due to the high latency. These various approaches each investigate the trade-off between the expressiveness of reasoning and the efficiency of processing [5].

RDF Stream Processing (RSP) identifies a family of CP engines that solve information needs over heterogeneous streaming data, which is typical in IoT applications. It addresses data variety by adopting RDF streams as data model, and solves data velocity by extending SPARQL with the continuous semantics [16]. Different RSP engines exist, such as C-SPARQL [17], CQELS [18], Yasper [19] and RSP4J [20]. Queries can be registered to these engines that are used to continuously filter the defined data streams. A data window is placed on top of the data stream. Parameters of the window definition include the size of the data window that is added to the query's data model, and the window's sliding step which directly influences the query's evaluation frequency.

RSP-QL [21] is a reference model that unifies the semantics of the existing RSP approaches. RSP has been extended to support ROT in various ways: (i) solutions incorporating efficient incremental maintenance of materializations of the windowed ontology streams [22–25], (ii) solutions for expressive Description Logics (DL) [26, 27], and (iii) a solution for Answer Set Programming (ASP) [28]. More central to ROT is the logic-based framework for analyzing reasoning over streams (LARS) [29] that extends ASP for analytical reasoning over data streams. LASER [30] is a system, based on LARS, that employs a tractable fragment of LARS that ensures uniqueness of models. BigSR [31] employs Big Data technologies (e.g., Apache Spark and Flink) to evaluate the positive fragment of LARS. C-Sprite [32] focuses on efficient hierarchical reasoning to improve the throughput and application on edge devices by efficiently filtering out unnecessary data in the stream. A similar approach to filter out unnecessary streaming data in ASP exists, by investigating the dependency graph of the input

data [33]. RDF Event Processing (RSEP) identifies a family of approaches that extend CP over RDF Streams with event pattern matching [34]. RSEP extends RSP with a reactive RAT formalism with limited expressiveness [35]. RSEP-QL [36] is an extension of RSP-QL that incorporates the language features from Complex Event Processing (CEP) [37]. StreamQR [38] rewrites continuous RSP queries to multiple parallel queries, allowing for the support of ontologies that are expressed in the $\mathcal{ELHIO}$ logic. The CityPulse project [39] presents the combination of RSP, CEP and expressive reasoning through ASP.

The most advanced attempts to develop expressive Stream Reasoning increased the reasoning expressiveness, but at the cost of limited efficiency. DyKnow [40] and ETALIS [41] combine RAT and ROT reasoning, but perform CP at an extremely slow speed. STARQL [42] is a first step in the right direction because it mixes RAT, and ROT reasoning utilizing a Virtual Knowledge Graph (VKG) approach [43] to obtain CP. Cascading Reasoning [44] was proposed to solve the problem of expressive reasoning over high-frequency streams by introducing a hierarchical approach consisting of multiple layers. Although several of the presented approaches adopt a hierarchical approach [28, 41, 42], only a recent attempt has laid the first fundamentals on realizing the full vision of cascading reasoning with Streaming MASSIF [45].

3.2.2 Semantic IoT platforms and privacy preservation

Today, different IoT platforms exist that extend big data platforms with IoT integrators [46, 47]. FIWARE [48] is a platform that offers different APIs that can be used to deploy IoT applications. Sofia2 [49] is a semantic middleware platform that allows different systems and devices to become interoperable for smart IoT applications. SymbIoTe [50] goes a step further and abstracts existing IoT platforms by providing a virtual IoT environment provisioned over various cloud-based IoT platforms. The Agile [51] and BIG IoT [52] platforms focus on flexible IoT APIs and gateway architectures, such as VICINITY [53] and INTER-IoT [54] which also provide an interoperability platform. bIoTope [55] addresses the requirement for open platforms within IoT systems development.

Zooming in on IoT-based healthcare systems, a large number of solutions have risen in the last few years [1, 56–58]. Jaiswal et al. surveyed 146 healthcare for IoT solutions in recent years, and classified them in five categories: sensor-based, resource-based, communication-based, application-based, and security-based approaches. They identified scalability and interoperability as two big challenges that are yet to be solved by many systems. Especially the latter is a challenge with the heterogeneity of data originating from different sources. This challenge can be solved with Semantic Web technologies.

Focusing on IoT healthcare systems that involve semantic technologies, multiple solutions already exist. For example, in the topic of homecare monitoring, Zgheib et

al. [59] proposed a scalable semantic framework to monitor activities of daily living in elderly, to detect diseases and epidemics. The proposed framework is based on several semantic reasoning techniques that are distributed over a semantic middleware layer. It makes use of CEP to extract symptom indicators, which are fed to a SPARQL engine that detects individual diseases. C-SPARQL is then employed on a stream of diseases to detect possible epidemics. While this approach zooms in largely on scalability for this specific use case, it does not offer any flexibility in making the SPARQL and C-SPARQL queries context-aware in a fully automated and adaptive way.

Moreover, Jabbar et al. [60] and Ullah et al. [61] both presented an IoT-based Semantic Interoperability Model that provides interoperability among heterogeneous IoT devices in the healthcare domain. These models add semantic annotations to the IoT data, allowing SPARQL queries to easily extract concepts of interest. However, these illustrative SPARQL queries require manual configuration effort and are not automatically ensuring context-awareness in a dynamic environment. In addition, Ali et al. [62] present an ontology-aided recommendation system to efficiently monitor the patient's physiology based on wearable sensor data while recommending specific, personalized diets. Similarly, Subramaniyaswamy et al. [63] present a personalized travel and food recommendation system based on real-time IoT data about the patient's physical conditions and activities. Again, these systems only work with static SPARQL queries to evaluate their system, not achieving context-awareness in an adaptive, dynamic environment.

In summary, many of the presented platforms are adopting a wide range of existing Semantic Web technologies to deal with the challenges associated with real-time IoT applications in complex IoT domains such as healthcare. These platforms typically combine different technologies that involve both stream processing and semantic reasoning components. They all have in common that the queries for the stream processing components are not yet configured and managed in a fully automated, adaptive and context-aware way.

Privacy by design is an approach that states that privacy must be incorporated into networked data systems and technologies, by default [8, 64, 65]. It approaches privacy from the design-thinking perspective, stating that the data controller of a system must implement technical measures for data regulation by default, within the applicable context. Privacy by design is a broad concept that is more concretely defined through seven principles that can be applied to the design of a system. One of these principles is that the privacy-preserving capabilities should be embedded into the design and architecture of IT systems. Another principle focuses on the importance of keeping privacy user-centric, ensuring that the design always considers the needs and interests of the users. Other principles focus on visibility and transparency, privacy as the default setting, proactive instead of reactive measures, avoiding unnecessary privacy-related trade-offs, and end-to-end security through the life cycle of

the data. Privacy by design is a key principle of the General Data Protection Regulation (GDPR) of the European Union [65].

3.3 Use case description and set-up

To demonstrate how DIVIDE can be employed in a semantic IoT network to perform context-aware homecare monitoring, a detailed use case is presented in this section.

3.3.1 Use case description

The homecare monitoring use case scenario presented in this chapter focuses on a rule-based service that recognizes the activities of elderly people in their homes.

Use case background More and more people live with chronic illnesses and are followed up at home by various healthcare actors such as their General Practitioner (GP), nursing organization, and volunteers. Patients in homecare are increasingly equipped with monitoring devices such as lifestyle monitoring devices, medical sensors, localization tags, etc. The shift to homecare makes it important to continuously assess whether an alarming situation occurs at the patient. If an alarm is generated, either automatically or initiated by the patient, a call operator at an alarm center should decide which intervention strategy is required. By reasoning on the measured parameters in combination with the medical domain knowledge, a system could help a human operator with choosing the most optimal intervention strategy.

A core building block of a homecare monitoring solution is an autonomous activity recognition (AR) service that detects and recognizes different in-home activities performed by the patient. Moreover, it should also monitor whether ongoing activities belong to a known regular routine of the patient, so that anomalies in the patient's daily activity pattern can be detected. Such a service could make use of the data collected by the different sensors and devices installed in the patient's home environment, as well as knowledge about AR rules and known routines of the patient. Given the heterogeneous nature of these different data sources, Semantic Web technologies are ideally suited to create this autonomous AR service.

Details of the activity recognition service The use case of routine and non-routine AR has been designed together with the home monitoring company Z-Plus. To properly perform knowledge-driven AR, AR rules should be known by the system. Z-Plus helped us with designing the rules.

An AR rule can be defined as a set of one or more value conditions defined on certain observable properties that are being analyzed for a certain entity type. An observable property is any property that can be measured by a sensor in the patient's environment, e.g., temperature, relative humidity, power consumption, door status (open

vs. closed), indoor location, etc. Every sensor analyzes its property for a specific entity. Examples of analyzed entities are a room (e.g., for a humidity sensor), an electrical appliance such as a cooking stove (e.g., for a power consumption sensor), a cupboard (e.g., for a door contact sensor), or even the patient (e.g., for a wearable sensor).

In a realistic home environment with a wide range of sensors installed, many different AR rules will be defined. This makes it highly inefficient to continuously monitor all possible activities that can be recognized in the home, since this would require the continuous monitoring of all sensors that observe a certain property for an entity type associated with at least one rule. Hence, the AR service performs location-dependent activity monitoring: it only observes activities that are relevant to the room that the patient is currently located in. To enable this, an indoor location system should be installed that unambiguously knows the current room of the patient at every point in time. The activities relevant to the current room can be derived by considering all sensors that analyze this room or an entity in the room: all activity rules should be evaluated that have conditions (i) on observable properties that are measured by these sensors, and (ii) that are defined for the same entity type as analyzed by those sensors.

Activities recognized by the AR service should be labeled as belonging to the regular routine of this patient or not. If an ongoing activity in the patient's routine is recognized, the situation is normal and requires no more strict follow-up. Ideally, as long as an activity is going on, location changes in the home are less probable and should therefore be monitored less frequently. However, if an activity outside the routine of the patient is being detected, more strict location monitoring is required since the situation is abnormal. If necessary, an alarm should automatically be generated by the system. To implement such a system, knowledge on the existing routines of the patient at different times of the day should exist.

Finally, an important requirement of the AR service is that it reduces the information that leaves the patient's home environment to a minimum, as a first step in preserving the patient's privacy. This implies that no actual raw sensor data should be sent over the network. To enable this, the AR service should largely run in-home, so that only the actual outputs such as detected activities are being sent. Obviously, data that is not contained in the HomeLab should always be sent over a secure, encrypted connection.

Running example To facilitate the methodological description of DIVIDE in Sections 3.4, 3.5 and 3.6, consider the following illustrative running example derived from the presented homecare monitoring use case.

> Consider a smart home with an indoor location system detecting in which room the patient is present, and an environmental sensor system measuring the relative humidity in every room of the home. The smart home consists of multiple rooms including one bathroom. The patient living in the home has a

morning routine that includes showering. To keep it simple, the AR service of the running example consists of a single rule. This rule detects when a person is showering, and is formulated as follows:

A person is showering if the person is present in a bathroom with a relative humidity of at least 57%.

This is a rule with a single condition, defined on a crossed lower threshold for the relative humidity observable property, for the bathroom entity type. Hence, given the presence of a humidity sensor in the patient's bathroom, the showering activity will be monitored by the AR service *if* the patient is located in the bathroom.

3.3.2 Activity recognition ontology

An Activity Recognition ontology has been designed to support the described use case scenario. This Activity Recognition ontology is linked to the DAHCC (Data Analytics for Health and Connected Care) ontology [66], which is an in-house designed ontology that includes different modules connecting data analytics to healthcare knowledge. Specifically for the purpose of this semantic use case, it is extended with a module `KBActivityRecognition` supporting the knowledge-driven recognition of in-home activities.

The DAHCC ontology contains five main modules. The `SensorsAndActuators` and `SensorsAndWearables` modules describe the concepts that allow defining the observed properties, location, observations and/or actions of different sensors, wearables and actuators in a monitored environment such as a smart patient home. The `MonitoredPerson` and `CareGiver` modules contain concepts for the definition of a patient monitored inside a residence and the patient's caregivers. The `ActivityRecognition` module allows describing the activities performed by a monitored person that are predicted by an AR model.

The DAHCC ontology bridges the concepts of multiple existing ontologies in the data analytics and healthcare domains. These ontologies include SAREF (the Smart Applications REFerence ontology) [67] and its extensions SAREF4EHAW (SAREF extended with concepts of the eHealth Ageing Well domain) [68], SAREF4BLDG (an extension for buildings and building spaces) and SAREF4WEAR (an extension for wearables), as well as the Execution-Executor-Procedure (EEP) ontology [69].

Listing 3.1 shows how a knowledge-based AR model can be defined and configured. In the example, it is configured according to the use case's running example, i.e., with one activity rule for showering. Lines 13–17 of this listing contain the definition of the single condition of this rule.

In Section 3.A.1 of Addendum 3.A, additional listings detail multiple other definitions within the Activity Recognition ontology that support the knowledge-driven AR

Listing 3.1: Example of how a knowledge-based AR model with an activity rule for showering can be described through triples in the `KBActivityRecognition` ontology module. All definitions are listed in RDF/Turtle syntax. To improve readability, the `KBActivityRecognition:` prefix is replaced by the `:` prefix.

```
# define knowledge-based activity recognition model and its config with a specific rule
:KBActivityRecognitionModel rdf:type ActivityRecognition:ActivityRecognitionModel ;
    eep:implements :KBActivityRecognitionModelConfig1 .
:KBActivityRecognitionModelConfig1 rdf:type ActivityRecognition:Configuration ;
    :containsRule :showering_rule .

# define showering activity rule: detected by one specific condition
:showering_rule rdf:type :ActivityRule ;
    ActivityRecognition:forActivity [ rdf:type ActivityRecognition:Showering ] ;
    :hasCondition :showering_condition .

# define showering condition: relative humidity in the bathroom should be at least 57%
:showering_condition rdf:type :RegularThreshold ;
    :forProperty [ rdf:type SensorsAndActuators:RelativeHumidity ] ;
    Sensors:analyseStateOf [ rdf:type SensorsAndActuators:BathRoom ] ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "57"^^xsd:float .

# define in system that conditions can be defined on relative humidity in a room
SensorsAndActuators:RelativeHumidity rdfs:subClassOf :ConditionableProperty .
SensorsAndActuators:Room rdfs:subClassOf :AnalyzableForCondition .
```

use case and its running example. This includes the ontological definitions that can be used by a semantic reasoner to define whether an activity prediction corresponds to a person's routine, as well as the semantic description of the example patient and home in the running use case example.

3.3.3 Architectural use case set-up

To implement the use case scenario of a knowledge-driven routine and non-routine AR service, a cascading reasoning architecture is used [70]. An overview of the architectural cascading reasoning set-up for this use case is shown in Figure 3.1. This architecture is generic and can be applied to different use case scenarios in the healthcare domain with similar requirements.

The architecture of the system is split up in a local and a central part. The local part consists of a set of components that are running on a local device in the patient's environment. This device could be any existing gateway that is already installed in the patient's home, such as the device for a deployed nurse call system. The local components are the Semantic Mapper and an RSP Engine. The components of the central part are deployed on a back-end server of an associated nursing home or hospital. They consist of a Central Reasoner, DIVIDE, and a Knowledge Base.

Knowledge Base The Knowledge Base contains the semantic representation of all domain knowledge and context data in the system, in an RDF-based knowledge graph. In the given use case scenario, this domain knowledge consists of the Activity

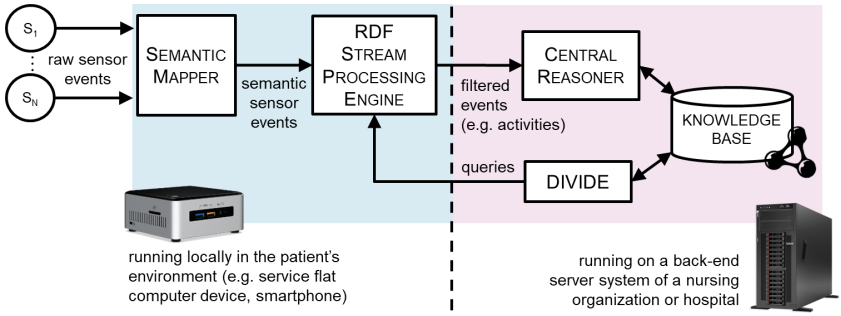


Figure 3.1: Architectural set-up of the eHealth use case scenario

Recognition ontology that is discussed in Section 3.3.2. It includes the AR model with its activity rules. The contextual information describes the different smart homes and their installed sensors, and patients.

Semantic Mapper The Semantic Mapper semantically annotates all raw observations generated by the sensors in the patient's environment. These semantic sensor observations are forwarded to the data streams of the RSP Engine.

RSP Engine The RSP engine continuously evaluates the registered queries on the RDF data streams, to filter relevant events. In this use case scenario, the filtered events are in-home locations and recognized activities both in and not in the patient's routine. Only these filtered events are encrypted and sent over the network to the Central Reasoner. By applying the cascading reasoning principles and installing the RSP Engine locally in the patient's environment, a first step in preserving the patient's privacy can be taken.

Central Reasoner The Central Reasoner is responsible for further processing the events received from the RSP Engine, and acting upon them. For example, it can aggregate the filtered events and save them to use for future call enrichment, or send an alarm to the patient's caregivers when necessary. In general, any action is possible, depending on what additional components are deployed and implemented on the central node.

Importantly, the Central Reasoner will also update relevant contextual information in the Knowledge Base, such as events occurring in the patients' environment. This information can then trigger a re-evaluation of the queries deployed on the local RSP engines. In the given use case scenario, relevant context changes that trigger a possible change in the deployed RSP queries are location updates and detected activities. When the in-home location of the patient changes, the set of activities that need to be monitored changes as well, since the AR service is location-dependent. Moreover,

context information about recognized ongoing routine and non-routine activities directly defines the execution frequency of the location monitoring RSP query.

DIVIDE DIVIDE is the component that manages the queries executed by the local RSP Engine components. It updates the queries whenever triggered by context updates in the Knowledge Base. By aggregating contextual information with medical domain knowledge through semantic reasoning during the query derivation, the resulting RSP queries only involve filtering and do not require any more real-time reasoning. Moreover, it allows to dynamically manage the window parameters of the queries (i.e., the size of the data window and its sliding step) based on the current context. It is fully automated and adaptive, so that at all times, relevant queries are being executed given the context information about the patients in the Knowledge Base.

In the running example, DIVIDE will ensure that there is always a location monitoring query running on the RSP Engine component installed in the patient's home. The window parameters of this query will depend on whether or not an activity is currently going on, and whether or not this activity belongs to the current patient's routine. In addition, when the patient is located in the bathroom, an additional RSP query will be derived and installed that monitors when the patient is showering. When the query detects this activity, this would be considered a recognized routine activity as showering is included in the patient's morning routine.

3.4 Overview of the DIVIDE system

In Section 3.3, the general cascading reasoning architecture of the semantic system in the eHealth use case scenario is explained. This section zooms in on DIVIDE, the architectural component responsible for managing the queries running on the local RSP Engine components. It is the task of the DIVIDE system to ensure that these queries perform the relevant filtering given the current context, at any given time, for every RSP Engine known to DIVIDE.

The methodological design of DIVIDE contains of two main pillars: (i) the initialization of DIVIDE, involving the DIVIDE query parsing and ontology preprocessing steps, and (ii) the core of DIVIDE which is the query derivation. Figure 3.2 shows a schematic overview of the action steps, inputs and internal assets DIVIDE, in which the two main pillars can be distinguished. The following two sections, Section 3.5 and Section 3.6, provide more information on this initialization and query derivation, respectively. Throughout the descriptions of DIVIDE in these sections, the running eHealth use case example described in Section 3.3.1 is considered.

In terms of logic, DIVIDE works with the rule-based Notation3 Logic (N3) [14]. The semantic reasoner used within DIVIDE should thus be a reasoner supporting N3. Such a reasoner can reason within the OWL 2 RL profile [10], which implies that a semantic system that uses DIVIDE in combination with an RSP engine is equivalent

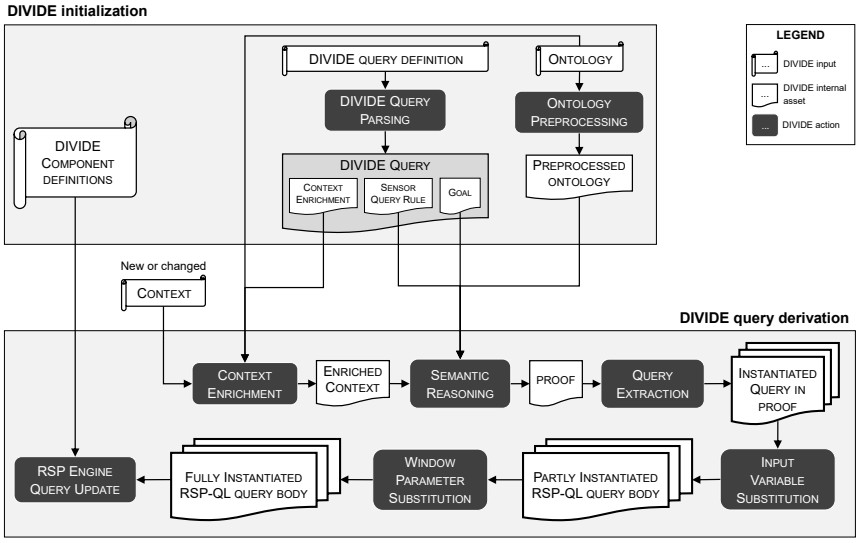


Figure 3.2: Schematic overview of the DIVIDE system. It shows all actions that can be performed by DIVIDE with their inputs and outputs. A distinction is made between internal assets and external inputs to the system. The overview is split up in the two major parts: the inputs, steps and assets of the DIVIDE initialization, and those of the DIVIDE query derivation.

to a set-up involving a semantic OWL 2 RL reasoner. The reasoner should support the generation of all triples based on a set of input triples and rules, as well as generating a proof towards a certain goal rule. Such a proof should contain the chain of all rules used by the reasoner to infer new triples based on its inputs, described in N3 logic.

3.5 Initialization of the DIVIDE system

The core task of DIVIDE is the derivation and management of the queries running on the RSP engines of the semantic components in the system that are known to DIVIDE. To allow DIVIDE to effectively and efficiently perform the query derivation for one or more components upon context changes, different initialization steps are required. Three main steps can be distinguished from the upper part of the DIVIDE system overview in Figure 3.2: (i) parsing and initializing the DIVIDE queries, (ii) preprocessing the system ontology, and (iii) initializing the DIVIDE components. This section zooms in on each of these three initialization tasks.

3.5.1 Initialization of the DIVIDE queries

A DIVIDE query is a generic definition of an RSP query that should perform a real-time processing task on the RDF data streams generated by the different local components in the system. The goal of DIVIDE is to instantiate this query in such a way that it can perform this task in a single query that simply filters the RDF data streams. To this end, the internal representation of a DIVIDE query contains a goal, a sensor query rule with a generic query pattern, and a context enrichment. These three items are essential for correctly deriving the relevant queries during the query derivation process. They will each be explained in detail in the first three subsections of this section.

In the running example, there is one RSP query that actively monitors the location of the patient in the home, and one query that detects a showering activity when the patient is located in the bathroom. This subsection will focus on the latter, which is an example of an actual AR query. Within DIVIDE, a generic DIVIDE query will be defined for each *type* of activity rule present in the system. This means that no dedicated DIVIDE query per activity should be defined, which would be too cumbersome and highly impractical in a real-world deployment. A rule type is a specific combination of conditions and the type of value they are defined on. For the showering rule, this means that the type is defined as follows: a rule with a single condition on a lower regular threshold that should be crossed. This means that the detailed specific RSP queries corresponding to activity rules of the same type will all be derived from the same generic DIVIDE query. The generic DIVIDE query corresponding to the type of the showering activity rule will be used as the running example DIVIDE query in this section. Note that the running example will only focus on the detection of this activity *in* the patient's routine.

3.5.1.1 Goal

The goal of a DIVIDE query defines the semantic output that should be filtered by the resulting RSP query. This required query output is translated to a valid N3 rule. This rule is used in the DIVIDE query derivation to ensure that the resulting RSP query is filtering this required RSP query output.

For the generic query definition corresponding to the RSP query that detects the showering activity in the running example, the goal is specified in Listing 3.2. It is looking for any instance of a `RoutineActivityPrediction`.

3.5.1.2 Sensor query rule with generic query pattern

The sensor query rule is the core of the DIVIDE query definition. It is a complex N3 rule that defines the generic pattern of the RSP query, together with semantic

Listing 3.2: Goal of the generic DIVIDE query detecting an ongoing activity in a patient's routine

```
{
    ?p rdf:type KBActivityRecognition:RoutineActivityPrediction ;
       ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
       ActivityRecognition:activityPredictionMadeFor ?patient ;
       ActivityRecognition:predictedBy ?model ; saref-core:hasTimestamp ?t .
    ?activityType rdfs:subClassOf KBActivityRecognition:DetectableActivity .
} => {
    _:p rdf:type KBActivityRecognition:RoutineActivityPrediction ;
        ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
        ActivityRecognition:activityPredictionMadeFor ?patient ;
        ActivityRecognition:predictedBy ?model ; saref-core:hasTimestamp ?t .
} .
```

information on when and how to instantiate it. Its usage by the semantic rule reasoner during the DIVIDE query generation defines whether or not this generic query should be instantiated given the involved context.

The formalism of the sensor query rule builds further on SENSdesc, which is the result of previous research [71]. This theoretical work was the first step in designing a format that describes an RSP query in a generic way that can be combined with formal reasoning to obtain the relevant queries that filter patterns of interest. By generalizing this format and integrating it into DIVIDE, it has become practically usable.

Each sensor query rule consists of three main parts: the relevant context in the rule's antecedence, and the generic query and ontology consequences defined in the rule's consequence.

Relevant context In the antecedence of the sensor query, the context in which the generic RSP query might become relevant is generically described. For each set of query variables for which the antecedence is valid, there is a chance that the rule, instantiated with these query variables, will appear in the proof constructed by the semantic reasoner during the query derivation. If this is the case, the query will be instantiated for this set of variables.

To explain the different parts, consider the DIVIDE query corresponding to the running example detecting the showering activity. Listing 3.3 defines the sensor query rule for the corresponding type of activity rule. The rule's antecedence with the relevant context of the sensor query rule is described in lines 2–24. In short, it looks for AR rules relevant to the current room of the patient, following the definition of location-dependent activity monitoring in Section 3.3.1.

Generic query The generic query definition is contained inside the consequence of the sensor query rule. It consists of three main aspects: the generic query pattern, its input variables, and its static window parameters.

Listing 3.3: Sensor query rule of the generic DIVIDE query detecting an ongoing activity in a patient's routine, for an activity rule type with a single condition on a certain property of which a value should cross a lower threshold (part 1/2)

```
{
    ?model rdf:type ActivityRecognition:ActivityRecognitionModel ;
        <https://w3id.org/eep#implements> [ rdf:type ActivityRecognition:Configuration ;
                                            KBActivityRecognition:containsRule ?a ] .
    ?a rdf:type KBActivityRecognition:ActivityRule ;
       ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
       KBActivityRecognition:hasCondition [
           rdf:type KBActivityRecognition:RegularThreshold ;
           KBActivityRecognition:isMinimumThreshold "true"^^xsd:boolean ;
           saref-core:hasValue ?threshold ;
           Sensors:analyseStateOf [ rdf:type ?analyzed ] ;
           KBActivityRecognition:forProperty [ rdf:type ?prop ]
       ] .

    ?activityType rdfs:subClassOf KBActivityRecognition:DetectableActivity .

    ?sensor rdf:type saref-core:Device ; saref-core:measuresProperty ?prop_o ;
            Sensors:isRelevantTo ?room ; Sensors:analyseStateOf [ rdf:type ?analyzed ] .
    ?prop_o rdf:type ?prop .

    ?prop rdfs:subClassOf KBActivityRecognition:ConditionableProperty .
    ?analyzed rdfs:subClassOf KBActivityRecognition:AnalyzableForCondition .

    ?patient MonitoredPerson:hasIndoorLocation ?room .
}
=>
{
    _:q rdf:type sd:Query ;
        sd:pattern sd-query:pattern ;
        sd:inputVariables (("?sensor" ?sensor) ("?threshold" ?threshold) "?activityType"
            ?activityType) ("?patient" ?patient) ("?model" ?model) ("?prop_o" ?prop_o)) ;
        sd:windowParameters (("?range" 30 time:seconds) ("?slide" 10 time:seconds)) .

    _:p rdf:type ActivityRecognition:ActivityPrediction ;
        ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
        ActivityRecognition:activityPredictionMadeFor ?patient ;
        ActivityRecognition:predictedBy ?model ; saref-core:hasTimestamp _:t .
} .

sd-query:pattern rdf:type sd:QueryPattern ;
    sh:prefixes sd-query:prefixes-activity-showering ;
    sh:construct """
        CONSTRUCT {
            _:p a KBActivityRecognition:RoutineActivityPrediction ;
                ActivityRecognition:forActivity [ a ?activityType ] ;
                ActivityRecognition:activityPredictionMadeFor ?patient ;
                ActivityRecognition:predictedBy ?model ; saref-core:hasTimestamp ?now .
        }
        FROM NAMED WINDOW :win ON <http://protego.ilabt.imec.be/idlab.homelab> [RANGE ?{
              range} STEP ?{slide}]
        WHERE {
            BIND (NOW() as ?now)
            WINDOW :win { ?sensor saref-core:makesMeasurement [
                                saref-core:hasValue ?v ; saref-core:hasTimestamp ?t ;
                                saref-core:relatesToProperty ?prop_o ] .
                          FILTER (xsd:float(?v) > xsd:float(?threshold)) }
        }
        ORDER BY DESC(?t) LIMIT 1""" .

```

Listing 3.3: Sensor query rule of the generic DIVIDE query detecting an ongoing activity in a patient's routine, for an activity rule type with a single condition on a certain property of which a value should cross a lower threshold (part 2/2)

```
59  sd-query:prefixes-activity-showering rdf:type owl:Ontology ;
60      sh:declare [ sh:prefix "xsd" ;
61                   sh:namespace "http://www.w3.org/2001/XMLSchema#"^^xsd:anyURI ] ;
62      sh:declare [ sh:prefix "saref-core" ;
63                   sh:namespace "https://saref.etsi.org/core/"^^xsd:anyURI ] ;
64      sh:declare [ sh:prefix "ActivityRecognition" ;
65                   sh:namespace "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition
                          /"^^xsd:anyURI ] ;
66      sh:declare [ sh:prefix "KBActivityRecognition" ;
67                   sh:namespace "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/
                          KBActivityRecognition/"^^xsd:anyURI ] .
```

The generic query pattern is a string representation of the actual RSP-QL query that will be the result of the DIVIDE query derivation. This pattern is however still generic: some of its query variables still need to be substituted by actual values to obtain the correct and valid RSP-QL query. Similarly, the window parameters of the input stream windows of the RSP-QL query also need to be substituted.

The input variables that need to be substituted by the semantic reasoner in the generic query pattern are defined as a N3 list. Every item in this list represents one input variable. This input variable is a list itself as well: the first item represents the string literal of the variable in the generic query pattern to be substituted, the second item is the query variable that should occur in the sensor query rule's antecedence so that it is instantiated by the semantic reasoner if the rule is applied in the proof during the query derivation.

Similarly, the definition of the static window parameters is also a list of lists. Static window parameters are variables that should also be substituted by the semantic reasoner during the query derivation, but in the stream window definition instead of the query body or output. They are static as their value is directly defined by the value of the corresponding variable. Every item of the outer list is an inner list of three items. The first item represents the string literal of the variable in *a* window definition of the generic query pattern. The second item can either be a query variable or literal defining the value of the window parameter. If this is a query variable, it will be substituted during the rule evaluation based on the matching value in the rule's antecedence, similarly to the input variables. The third item defines the unit of the value.

In Listing 3.3, the generic query definition is described in lines 28–32 and lines 40–67. More specifically, lines 28–29 and lines 40–67 define the generic query pattern, whereas lines 30–31 and line 32 define the input variables and static window parameters of the generic query, respectively.

Inspecting the example in Listing 3.3 in further detail, the generic RSP-QL query pattern string is defined in lines 43–57. The query filters observations on the defined stream data window `:win` of a certain sensor `?sensor` with a value for the observed property `?prop_o` that is higher than a certain threshold `?threshold` (WHERE clause in lines 50–56). For every match of this pattern, output triples are constructed that represent an ongoing activity of type `?activityType` in the routine of a patient `?patient`, predicted by the activity recognition model `?model` (CONSTRUCT clause in lines 43–48). These six variables are exactly the six input variables as defined in lines 30–31: their values will be instantiated during the query derivation. Note that the window parameter definitions specified in line 32 of Listing 3.3 define a window size of 30 seconds and a window sliding step of 10 seconds.

Ontology consequences The ontology consequences are the second main part of the sensor query rule's consequence. This part describes the *direct* effect of a query result in a real-time reasoning context. This effect is obtained when a stream window of the generic RSP query would fulfill the pattern of the WHERE clause but no additional reasoning has been done (yet) to know the *indirect* consequences of this matching pattern. This is an essential aspect to understand: the purpose of DIVIDE is to derive queries that can make conclusions that are valid with the given context, through a single RSP-QL query without any reasoning involved. In a context without DIVIDE, these same *indirect* conclusions could only be made by performing an additional semantic reasoning step, based on the *direct* conclusions that are directly known from the matching query pattern. In other words, the triples defining the ontology consequences *can* be the same as the output of the generic RSP-QL query and thus the consequence of the rule representing the DIVIDE query's goal. However, in practice, it will often require an additional semantic reasoning step to see whether the ontology consequences actually imply the output of the generic RSP-QL query.

In the running example, the *direct* consequences of a sensor observation matching the WHERE clause in lines 50–56 of Listing 3.3 would be the fact that an ongoing activity of the given type is detected for the given patient (lines 34–37). The *indirect* consequences represented by the definitions in the RSP-QL query output (lines 44–47) state that this is an activity *in the patient's routine*.

3.5.1.3 Context enrichment

Prior to the start of the query derivation with the semantic reasoner, the current context can still be enriched by executing one or more SPARQL queries on this context. The context enrichment of a DIVIDE query consists of this ordered set of valid SPARQL queries.

It is important to note that context-enriching queries are not only used to add general context to the model, but also for the dynamic window parameter substitution as will be explained in Section 3.6.5.

For the running example, no context-enriching queries are part of the DIVIDE query definition. However, Addendum 3.A discusses the definition of a related DIVIDE query that does include a context enrichment.

3.5.1.4 DIVIDE query parser

As an end user of DIVIDE, it is not required to define a DIVIDE query according to its internal representation to properly initialize DIVIDE. Instead, the recommended way to define a DIVIDE query is by specifying an ordered collection of existing SPARQL queries that are applied in an existing rule-based stream reasoning system, or through an already existing RSP-QL query. Through DIVIDE, this set of ordered queries will be replaced in the semantic platform by a single RSP query that performs a semantically equivalent task. To enable this, DIVIDE contains a query parser, which converts such an external DIVIDE query definition its internal representation. The goal of this approach is to make it easy for an end user to integrate DIVIDE into an existing semantic (stream) reasoning system, without having to know the details of how DIVIDE works.

DIVIDE is applied in a cascading system architecture. It considers its equivalent regular (stream) reasoning system as a semantic reasoning engine in which the set of SPARQL queries is executed sequentially on a data model containing the ontology (TBox) triples and rules, context (ABox) triples, and triples representing the sensor observations in the data stream. Each query in the ordered collection, except for the final one, should be a CONSTRUCT query, and its outputs are added to the data model on which (incremental) rule reasoning is applied before the next query in the chain is executed.

The definition of a DIVIDE query as an ordered set of SPARQL queries includes a context enrichment with zero or more context-enriching queries, exactly one stream query, zero or more intermediate queries, and either no or exactly one final query. Besides these queries, such a DIVIDE query definition also includes a set of stream windows (required), a solution modifier (optional), and a variable mapping from stream query to final query (optional). The remainder of this subsection will discuss these different inputs in this DIVIDE query definition.

Stream query and context enrichment In the ordered set of SPARQL queries, it is important that there is exactly one query that reads from the stream(s) of sensor observations. This query is called the *stream query*. In some cases, this query will be the first in the chain. If this is not the case, any preceding queries are defined as

context-enriching queries in the DIVIDE query definition. Importantly, the WHERE clause conditions of the stream query should be part of named graphs defined as data inputs with a `FROM` clause, except for special SPARQL constructs such as a FILTER or BIND clause. The IRIs of the named graphs are used to distinguish which data is considered as part of the context, and which data will be put on the data stream. For the data streams, the named graph IRI should reflect the stream IRI. This stream IRI should also be defined as a stream window.

Final query The final query in the ordered set of SPARQL queries is called the *final query* in DIVIDE. A final query is optional: if it does not exist, the stream query is considered the final query.

Intermediate queries The intermediate queries are an ordered list of zero or more SPARQL queries. This list contains those queries in the original set of SPARQL queries that are executed between the stream and final query.

Stream windows Each data stream window that should be included as input in the resulting RSP-QL query should be explicitly defined. It consists of a stream IRI, a window definition, and a set of default window parameter values.

The stream IRI represents the IRI of the data stream. This IRI should exactly match the name of a named graph defined in the stream query. The window definition defines the specification of how the windows are created on the stream. *If* the user wants to define variable window parameters, named variables should be inserted into the places that will be instantiated during the query derivation. In DIVIDE, two types of variable window parameters exist: static and dynamic window parameters. Static window parameters *might* be substituted similarly to an input variable during the DIVIDE query derivation. Hence, the variable name of this window parameter should appear in the WHERE clause of the stream query, in a named graph that is *not* corresponding to a stream window. This will ensure that the variable name can be substituted as a regular input variable. During the DIVIDE query derivation, dynamic window parameters are substituted before static parameters. A dynamic window parameter can be defined in the output of a context-enriching query. In case no context-enriching query yields a value for the dynamic window parameter variable, the value of the static window parameter with the same variable name will be substituted. If no such static window parameter is defined, a default value will be used. Hence, for each such variable in the window definition that is not defined as a static window parameter, this default value should be defined by the end user.

Solution modifier If the resulting RSP-QL query should have a SPARQL solution modifier, this can be included in the DIVIDE query definition. Any unbound variable names in the solution modifier should be defined in a named graph of the stream query's WHERE clause that represents a stream window.

Variable mapping of stream to final query If a final query is specified, it often occurs that certain query variables in both the stream and final query actually refer to the same individuals. To make sure that DIVIDE parses the DIVIDE query input correctly, the mapping of these variable names should be explicitly defined. This is a manual required step. Often, they will have the same variable names, making this mapping trivial.

Parsing the end user definition of a DIVIDE query to its internal representation The DIVIDE query parser can construct the goal, sensor query rule and context enrichment of a DIVIDE query from its end user definition. The context enrichment requires no parsing, while the goal and sensor query rule are composed from the different inputs.

The goal of the DIVIDE query is directly constructed from the final query. If it is a CONSTRUCT query, the content of the WHERE clause is put in the antecedence of the goal, while the content of the CONSTRUCT clause represents the goal's consequence. For any other query form, the WHERE clause of the final query is used for both the goal's antecedence and consequence. If no final query is available, the antecedence and consequence of the goal are copied from the result of the stream query. If the stream query is no CONSTRUCT query, the SELECT, ASK or DESCRIBE result clause is first converted to a triple pattern containing all its unbound variables.

The sensor query rule is the most complex part to construct. In the standard case, disregarding any exceptions, the antecedence of the rule is composed from all named graph patterns in the WHERE clause of the stream query that do *not* represent a stream graph. The ontology consequences in the consequence of the sensor query rule are copied from the stream query's output. The generic RSP-QL query pattern is constructed from different parts. Its resulting CONSTRUCT, SELECT, ASK or WHERE clause is directly copied from the result clause of the final query, or the stream query if no final query is present. Its input stream window definitions are constructed using the defined stream windows. The WHERE clause contains the content of the stream graphs in the stream query's WHERE clause, and the special SPARQL patterns that are not put inside a named graph pattern. If a solution modifier is specified, it is appended to the generic RSP-QL query pattern. The input variables and window parameters of the sensor query rule are derived by analyzing the stream query, final query and the variable mapping between both. Any intermediate queries are converted to additional semantic rules that are appended to the main sensor query rule.

Finally, it is worth noting that a DIVIDE query can alternatively also be defined through an existing RSP-QL query. Such a definition is quite similar to the definition described above, with a few differences. The main difference is that by definition, no intermediate and final queries will be present since the original system already uses

RDF stream processing and individual RSP-QL queries. This means no variable mapping should be defined either. Hence, this definition is typically more simple than the definition of a DIVIDE query as a set of SPARQL queries.

For the running use case example, the DIVIDE query that performs the monitoring of the showering activity rule can be defined as a set of ordered SPARQL queries. The DIVIDE query parser will translate this definition into the internal representation of this DIVIDE query, exactly as discussed in the previous subsections. This end user definition is discussed in detail in Section 3.A.2 of Addendum 3.A.

3.5.2 Initialization of the DIVIDE ontology

To properly perform the query derivation, an ontology should be specified as input to DIVIDE by the end user. During initialization, this ontology will be loaded into the system. By definition, this ontology is considered not to change often during the system's lifetime, in contrast with the context data. Therefore, the ontology should be preprocessed by the semantic reasoner wherever possible. This will speed up the actual query derivation process, since it avoids that the full ontology is loaded and processed every time the DIVIDE query derivation is triggered. To what extent the ontology can be preprocessed depends on the semantic reasoner used.

For the running example, the triples and axioms in the `KBActivityRecognition` module of the Activity Recognition ontology are preprocessed, including the definitions in all its imported ontologies.

3.5.3 Initialization of the DIVIDE components

To properly initialize DIVIDE, it should have knowledge about the components it is responsible for. A component is defined as an entity in the IoT network on which a single RSP engine runs. For each DIVIDE component, the following information should be specified by an end user for the correct initialization of DIVIDE:

- The name of the graph (ABox) pattern in the knowledge base that contains the context specific for the entity that this component's RSP engine is responsible for. A typical example in the eHealth scenario is a graph pattern of a specific patient, containing all patient information.
- A list of any additional graph patterns in the knowledge base that contain context relevant to the entity that this component's RSP engine is responsible for. An example is generic information on the layout of the environment in which the patient's smart home is situated. Such context

information is relevant to multiple components, and is therefore stored in separate graphs in the knowledge base.

- The type of the RSP engine of this component (e.g., C-SPARQL).
- The base URL of the RSP engine's server API. This API should support registering and unregistering RSP queries, and pausing and restarting an RSP stream. It will be used during the DIVIDE query derivation.

Upon initialization, all component information is processed and saved by DIVIDE. For every graph pattern associated with at least one component, DIVIDE should actively monitor for any updates to this ABox in the knowledge base, to trigger the query derivation for the relevant components when updates occur.

3.6 DIVIDE query derivation

Whenever DIVIDE is alerted of a context change in the knowledge base, the DIVIDE query derivation is triggered for every DIVIDE query. Based on the name of the updated ABox graph and the components known by the system, DIVIDE knows for which components the query derivation process should be started. This process can be executed independently, i.e., in parallel, for each combination of component and DIVIDE query. Hence, this section will focus on the query derivation task for a single component and a single DIVIDE query.

The DIVIDE query of the running example, that performs the monitoring of the showering activity rule, will be further used in this section to illustrate the query derivation process. The query derivation is triggered if any relevant context for a given component is updated. For this example, this context consists of all information about the patient and the smart home. Moreover, it also contains the output of the RSP queries: the in-home patient location and the detected ongoing activities.

The DIVIDE query derivation task for one RSP engine and one DIVIDE query consists of several steps, which are executed sequentially: (i) enriching the context, (ii) semantic reasoning on the enriched context to construct a proof containing the details of derived queries and how to instantiate them, (iii) extracting these derived queries from the proof, (iv) substituting the instantiated input variables in the generic RSP-QL query pattern for every derived query, (v) substituting the window parameters in a similar way, and (vi) updating the active RSP queries on the corresponding RSP engine. The input of the query derivation is the updated context, which consists of the set of triples in the context graph(s) of the knowledge base that are associated with the given component's RSP engine. In the following subsections, the DIVIDE query derivation

action steps are further detailed. Figure 3.2 shows a schematic overview of these steps on the bottom part. For every step, the inputs and outputs are detailed on the figure.

3.6.1 Context enrichment

Prior to actually deriving the RSP queries for the given DIVIDE query, the context data model can still be enriched. This is done by executing the ordered set of context-enriching queries corresponding to the DIVIDE query with a SPARQL query engine, if there are any, possibly after performing rule-based reasoning with the ontology axioms. The result of this step is a data model containing the original context triples and all triples in the output of any of the context-enriching queries, if there are any. Note that the output of the context-enriching queries can also contain dynamic window parameters to be used in the window parameter substitution step of the query derivation.

> The generic DIVIDE query corresponding to the running example of detecting the showering activity does not contain any context-enriching query. Hence, the updated context will directly be sent to the input of the next step. In Section 3.A.3 of Addendum 3.A, two additional examples are discussed of DIVIDE queries related to the running example that do contain context-enriching queries.

3.6.2 Semantic reasoning to derive queries

Starting from the enriched context data model, the semantic reasoner used within DIVIDE is run to perform the actual query derivation. This way, the reasoner will define whether the DIVIDE query should be initialized for the given context. If so, it specifies with what values the input variables and static window parameters, as defined in the query's sensor query rule consequence, should be substituted in the generic query pattern of the DIVIDE query.

The inputs of the semantic reasoner in this step consist of the preprocessed ontology (i.e., all triples and rules extracted from its axioms), the enriched context triples, the sensor query rule and the goal of the DIVIDE query. Given these inputs, the reasoner performs semantic reasoning to construct and output a proof with all possible rule chains in which the goal of the DIVIDE query is the final rule applied. Every such rule chain will be (partially) different and correspond to a different set of instantiated query variables appearing in the goal's rule.

To allow the semantic reasoner to construct a rule chain that starts from the context and ontology triples and ends with the goal rule, the sensor query rule is crucial. If the inputs allow the reasoner to derive the set of triples in the antecedence of the sensor query rule for a certain set of query variables, the rule *can* be evaluated for this set of variables. However, the semantic reasoner *will* only actually evaluate the rule for this set and include it in the rule chain, *if* the triples in the consequence of the sensor query rule (and more specifically, the part with the ontology consequences) allow

the semantic reasoner to derive the antecedence of the goal rule. This can be either directly (i.e., without semantic reasoning) or indirectly (i.e., after rule-based semantic reasoning). If this is not the case, the sensor query rule will not help the semantic reasoner in constructing a rule chain where the goal is the last rule applied, for the given set of sensor query rule variables. Hence, if the proof contains an instantiation of the sensor query rule for a given set of query variables, this implies that the generic RSP-QL query of this DIVIDE query should be instantiated for this set. This should be done with those query variables of this set that are present in the list of input variables or window parameters of the sensor query rule's consequence.

To reassure that this process works, consider the DIVIDE query parser's translation of the ordered set of SPARQL queries in the end user DIVIDE query definition into its internal representation. If the original stream query in the SPARQL input would yield a query result, the final query's WHERE clause *might* have a matching pattern, and thus an output. This is equivalent to the potential evaluation of the sensor query rule in the proof, depending on whether the sensor query rule's consequence directly or indirectly leads to a matching antecedence of the goal rule.

When the query derivation is executed for the DIVIDE query of the running example, the inputs will include the showering AR rule in Listing 3.1 that is defined in the preprocessed ontology. In the proof constructed by the semantic reasoner, the DIVIDE query's sensor query rule of Listing 3.3 would be instantiated once for the showering activity, *if* the current location of the patient is the bathroom. The step in the rule chain of the reasoner's proof in which this happens, is shown in Listing 3.4. This proof shows that the relative humidity sensor with the given ID can detect the showering activity for patient with ID 157 if its value is 57 or higher. If the current context would describe another patient location than the bathroom, or would not define showering as part of the routine of the patient with ID 157, the proof would not contain this sensor query rule instantiation.

3.6.3 Query extraction

The proof in the output of the semantic reasoning step *can* contain instantiations of the sensor query rule. If not, the proof will be empty, since this means that the semantic reasoner has not found any rule chain that leads to an instantiation of the goal rule. Every sensor query rule instantiation in the proof contains the list of input variables and window parameters that need to be substituted into the generic RSP-QL query of the considered DIVIDE query. In the query extraction step, DIVIDE will extract these definitions from every sensor query rule instantiation in the proof. Hence, the output of this step is a set of zero, one or more extracted queries.

Listing 3.4: One step of the proof constructed by the semantic reasoner used in DIVIDE during the DIVIDE query derivation for the generic DIVIDE query of the running use case example. It shows how the sensor query rule in Listing 3.3 is instantiated in the proof's rule chain. `[...]` is a placeholder for omitted parts that are not of interest.

```
@prefix r: <http://www.w3.org/2000/10/swap/reason#>.

<#lemma3> a r:Inference;
  r:gives {
    _:sk_0 a sd:Query.
    _:sk_0 sd:pattern sd-query:pattern.
    _:sk_0 sd:inputVariables (
      ("?sensor" <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78>)
      ("?threshold" "57"^^xsd:float)
      ("?activityType" ActivityRecognition:Showering)
      ("?patient" patients:patient157)
      ("?model" :KBActivityRecognitionModel)
      ("?prop_o" <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/org.dyamand.types.
            common.RelativeHumidity>)
    ).
    _:sk_0 sd:windowParameters (("?range" 30 time:seconds) ("?slide" 10 time:seconds)).
    _:sk_1 a ActivityRecognition:ActivityPrediction.
    _:sk_1 ActivityRecognition:forActivity _:sk_2.
    _:sk_2 a ActivityRecognition:Showering.
    _:sk_1 ActivityRecognition:activityPredictionMadeFor patients:patient157.
    _:sk_1 ActivityRecognition:predictedBy :KBActivityRecognitionModel.
    _:sk_1 saref-core:hasTimestamp _:sk_3.
  };
  r:evidence ( <#lemma8> [...] <#lemma31> );
  [...]
  r:rule <#lemma32>.
```

Listing 3.5: Output of the query extraction step of the DIVIDE query derivation, performed for the running example on the proof with a single sensor query rule instantiation presented in the proof step of Listing 3.4. The extraction of the dynamic window parameters (line 17) is done on the enriched context outputted by the context enrichment step.

```
1  @prefix : <file:///home/divide/.divide/query-derivation/10-10-129-31-8175-/activity-
       ongoing/20211220_194006/proof.n3#>.
2
3  # output of the first reasoning step of the query extraction
4  :lemma3 a sd:Query.
5  :lemma3 sd:inputVariables (
6      ("?sensor"
7       <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78>)
8      ("?threshold" "57"^^xsd:float) ("?activityType" ActivityRecognition:Showering)
9      ("?patient" patients:patient157) ("?model" :KBActivityRecognitionModel)
10     ("?prop_o" <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/org.dyamand.
           types.common.RelativeHumidity>)
11 ).
12 :lemma3 sd:staticWindowParameters (("?range" 30 time:seconds)
13                                    ("?slide" 10 time:seconds)).
14 :lemma3 sd:pattern sd-query:pattern.
15
16 # output of the second reasoning step of the query extraction
17 :lemma3 sd:dynamicWindowParameters ().
```

The query extraction happens through two forward reasoning steps with the semantic reasoner used in DIVIDE. The outputs of both steps are combined to construct the output of the query extraction. The first reasoning step extracts the relevant content from the sensor query rule instantiations in the proof. For each instantiation, this content includes the instantiated input variables and window parameters, as well as a reference to the query pattern in which they need to be substituted. The second forward reasoning step of the query extraction retrieves any defined window parameters from the enriched context that are associated with the instantiated RSP-QL query pattern. Such window parameters may have been added to the enriched context during the context enrichment step. They will be used as dynamic window parameters during the window parameter substitution, while the window parameters occurring in the sensor query rule instantiations are considered as static window parameters.

For the running example, the output of the extraction for the proof step in Listing 3.4 is presented in lines 4–14 of Listing 3.5. Line 17 of this listing presents the output of the second step. For this query example, there are no dynamic window parameters, which defaults the output of this second query extraction step to an empty list.

In Section 3.A.3 of Addendum 3.A, a related example is presented that does include dynamic window parameters.

3.6.4 Input variable substitution

In this step, DIVIDE substitutes the input variables of each query from the query extraction output into the associated RSP-QL query pattern. To achieve this, a collection of N3 rules have been defined that allow to substitute the input variables into the query body in a deterministic way. Moreover, they ensure that the substitution is correct for IRIs and literals of any data type. To perform the substitution, the semantic reasoner used in DIVIDE performs a forward reasoning step. The input of this reasoning step consists of the substitution rules, the output of the query extraction step and the query pattern of the considered DIVIDE query. For each query in the query extraction output, the output of this step consists of a set of triples that define the partially substituted RSP-QL query body.

The output of the input variable substitution step for the running example is presented in Listing 3.6. This substitution is performed using the generic RSP-QL query body referenced in the output of the query extraction in Listing 3.5. This query body is shown in Listing 3.3. In the output, lines 1–13 redefine the prefixes, which will be required in a further step to construct the full RSP-QL query. Line 16 shows the current state of the instantiated RSP-QL query body: input variables have already been substituted, but the window parameters still need to be substituted. The static and dynamic window parameters that will be used for substitution in the following step, are propagated from the output of the query extraction step (lines 19–21).

3.6.5 Window parameter substitution

In this step, the window parameters are also substituted in the partially instantiated queries to obtain the resulting RSP-QL query bodies. This is the final step that is performed by the semantic reasoner used in DIVIDE.

In general, DIVIDE offers the possibility to define the window parameters of derived RSP queries using semantic definitions. Currently, context-aware window parameters can be defined by an end user via the definition of a DIVIDE query. By separating the window parameter substitution from the other query derivation steps, DIVIDE offers the flexibility to trigger this substitution for other reasons than a context change. An example of this could be a device monitor observing that the resources of the device cannot handle the current query execution frequency.

Currently, to enable the substitution of *use case dependent* window parameters, DIVIDE makes the distinction between static and dynamic window parameters. For a static window parameter, the variable behaves as a regular input variable. This means that it should be defined in the consequence of a DIVIDE query's sensor query rule with a triple similar to the following one:

```
_:q sd:windowParameters (("?range" ?var time:seconds)) .
```

Listing 3.6: Output of the input variable substitution step of the DIVIDE query derivation, performed for the running example on the query extraction output presented in Listing 3.5. The substitution is done using the generic RSP-QL query body of the corresponding DIVIDE query presented in Listing 3.3.

```
sd-query:prefixes-activity-showering a owl:Ontology.
sd-query:prefixes-activity-showering sh:declare _:bn_1.
_:bn_1 sh:prefix "xsd".
_:bn_1 sh:namespace "http://www.w3.org/2001/XMLSchema#"^^xsd:anyURI.
sd-query:prefixes-activity-showering sh:declare _:bn_2.
_:bn_2 sh:prefix "saref-core".
_:bn_2 sh:namespace "https://saref.etsi.org/core/"^^xsd:anyURI.
sd-query:prefixes-activity-showering sh:declare _:bn_3.
_:bn_3 sh:prefix "ActivityRecognition".
_:bn_3 sh:namespace "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/"^^xsd:
      anyURI.
sd-query:prefixes-activity-showering sh:declare _:bn_4.
_:bn_4 sh:prefix "KBActivityRecognition".
_:bn_4 sh:namespace "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/
      KBActivityRecognition/"^^xsd:anyURI.

_:sk_20 a sd:Query.
_:sk_20 sd:queryBody " CONSTRUCT { _:p a KBActivityRecognition:RoutineActivityPrediction
      ; ActivityRecognition:forActivity [ a <https://dahcc.idlab.ugent.be/Ontology/
      ActivityRecognition/Showering> ] ; ActivityRecognition:activityPredictionMadeFor <
      http://protego.ilabt.imec.be/idlab.homelab/patients/patient157> ;
      ActivityRecognition:predictedBy <https://dahcc.idlab.ugent.be/Ontology/
      ActivityRecognition/KBActivityRecognition/KBActivityRecognitionModel> ; saref-core:
      hasTimestamp ?now . } FROM NAMED WINDOW :win ON <http://protego.ilabt.imec.be/idlab
      .homelab> [RANGE ?{range} STEP ?{slide}] WHERE { BIND (NOW() as ?now) WINDOW :win {
       <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78> saref
      -core:makesMeasurement [ saref-core:hasValue ?v ; saref-core:hasTimestamp ?t ;
      saref-core:relatesToProperty <https://dahcc.idlab.ugent.be/Homelab/
      SensorsAndActuators/org.dyamand.types.common.RelativeHumidity> ] . FILTER (xsd:
      float(?v) > xsd:float(\"57\"^^xsd:float)) } } ORDER BY DESC(?t) LIMIT 1 ".
_:sk_20 sh:prefixes sd-query:prefixes-activity-showering.
_:sk_20 sd:pattern sd-query:pattern.
_:sk_20 sd:staticWindowParameters (("?range" 30 time:seconds)
                                   ("?slide" 10 time:seconds)).
_:sk_20 sd:dynamicWindowParameters ().
```

This requires the variable `?var` to occur in the sensor query rule's antecedence. When defining a DIVIDE query as an end user using an ordered set of existing SPARQL queries, this can be achieved by ensuring that the variable name of this window parameter appears in the WHERE clause of the stream query, in a named graph that is not corresponding to a stream window. By definition, static window parameter variables will always receive a value in the query extraction output that *can* be used for substitution. In addition, dynamic window parameters are dynamically defined as triples in the outputs of context-enriching queries, similar to the following ones:

```
sd-query:pattern sd:windowParameters (
        [ sd-window:variable "range" ; sd-window:value 30 ;
          sd-window:type time:seconds ] ) .
```

Importantly, dynamic window parameters will *always* overwrite static ones. This means that during the window parameter substitution, dynamic window parameters will be substituted first. Next, static window parameters are substituted for those window parameter variables in the RSP-QL query body that have not yet been substituted.

The substitution order of static and dynamic window parameters implies a few important things. Multiple dynamic window parameters can be defined in different context-enriching queries of the same DIVIDE query, to handle different situations. It is however the responsibility of the end user that no more than one definition occurs for each window parameter variable in the enriched context. If multiple values are defined for the same window parameter variable, the one that is substituted will be chosen arbitrarily. If no value is defined for a window parameter variable in the enriched context either, the value of the static window parameter with the same variable name will be substituted. However, if no static window parameter value is defined for this variable either, the default value in the end user definition of the DIVIDE query will be substituted. To make this work, DIVIDE will define a window parameter in the sensor query rule of the DIVIDE query with the given default value, for each such variable.

> In the running example, the definition of the generic DIVIDE query associated with the detection of an ongoing showering activity does not contain any context-enriching query that defines a dynamic window parameter. However, Section 3.A.3 of Addendum 3.A discusses an example of a related DIVIDE query that does contain dynamic window parameter definitions in its context-enriching queries.

The actual substitution of window parameters is very similar to the input variable substitution. For both the static and dynamic window parameters, a forward reasoning step is performed with the semantic reasoner used in DIVIDE. The inputs of the reasoner are the output of the previous step and a collection of N3 rules that ensure the correct substitution in a deterministic way. The unit of the window parameter, which is either a valid XML Schema duration string or a time unit, defines how the window parameter value is exactly substituted in the query body string.

3.6.6 RSP engine query update

The output of the window parameter substitution step is a set of instantiated, valid RSP-QL queries that are contextually relevant for the given component. These queries are however still presented as a series of semantic triples. This final step constructs the actual RSP-QL query string, translates the query to the correct query language and updates the registered queries at the component's RSP engine.

Query construction This step constructs an actual RSP-QL query from the set of prefixes and the instantiated query body triples in the output of the window parameter substitution step.

For the running example, the RSP-QL query resulting from the query construction step is presented in Listing 3.7. This query is the result of performing the window parameter substitution and query construction on the output of the input variable substitution step presented in Listing 3.6.

Query translation The definition of a DIVIDE component contains the query language of its RSP engine. If this language differs from RSP-QL, e.g., C-SPARQL, the RSP-QL query is translated in this step to this other language.

Query registration update The output of the previous step is a set of translated RSP queries for the given DIVIDE query. Since the DIVIDE query derivation is triggered because of a detected context change relevant to this component, the queries on the RSP engine of this component should be updated to reflect this new situation. To do so, DIVIDE keeps track of the queries that are currently registered on the RSP engine for the given DIVIDE query. In this step, DIVIDE semantically compares the new set of instantiated translated RSP queries with this existing set of registered queries. Based on this comparison, any registered queries that are no longer in the new set of contextually relevant RSP queries are unregistered. New queries that are not running yet on the RSP engine, are registered.

For completeness, it is important to mention that during the full DIVIDE query derivation, the query processing on the RSP engine of the corresponding component should ideally be temporarily paused. This is to avoid that incorrect filtering is done, since DIVIDE already knows that the active queries might no longer be contextually relevant as soon as DIVIDE is informed of a context change for this component. During the pause, incoming observations on the RSP engine's streams should be buffered temporarily. This way, the queries can be restarted as soon as the RSP query update step finishes, and the buffered stream data can be fed to the RSP engine with their original timestamps.

Listing 3.7: Final RSP-QL query that is the result of performing the window parameter substitution and query construction steps of the DIVIDE query derivation, performed for the running example on the input variable substitution output presented in Listing 3.6

```
PREFIX ActivityRecognition: <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/>
PREFIX KBActivityRecognition:
    <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/KBActivityRecognition/>
PREFIX saref-core: <https://saref.etsi.org/core/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
    _:p a KBActivityRecognition:RoutineActivityPrediction ;
        ActivityRecognition:forActivity [
            a <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/Showering> ] ;
        ActivityRecognition:activityPredictionMadeFor
            <http://protego.ilabt.imec.be/idlab.homelab/patients/patient157> ;
        ActivityRecognition:predictedBy
            <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/KBActivityRecognition/
                 KBActivityRecognitionModel> ;
        saref-core:hasTimestamp ?now .
}
FROM NAMED WINDOW :win ON <http://protego.ilabt.imec.be/idlab.homelab> [RANGE PT30S STEP PT10S]
WHERE {
    BIND (NOW() as ?now)
    WINDOW :win {
        <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78>
            saref-core:makesMeasurement [
                saref-core:hasValue ?v ; saref-core:hasTimestamp ?t ;
                saref-core:relatesToProperty <https://dahcc.idlab.ugent.be/Homelab/
                     SensorsAndActuators/org.dyamand.types.common.RelativeHumidity> ] .
        FILTER (xsd:float(?v) > xsd:float("57"^^xsd:float))
    }
}
ORDER BY DESC(?t) LIMIT 1
```

3.7 Implementation of the DIVIDE system

The previous sections have described DIVIDE from a methodological point of view, irregardless of implementation details. This section zooms in on our implementation of DIVIDE.

3.7.1 Technologies

DIVIDE is implemented in Java as a set of Java JAR modules. These modules include the DIVIDE engine, which is the core of DIVIDE, the DIVIDE reasoning module and the DIVIDE server.

The DIVIDE reasoning module implements the ontology preprocessing and the query derivation steps with the semantic reasoner used in DIVIDE. Our implementation uses the EYE reasoner [72], which fulfills the requirements of the semantic reasoner explained in Section 3.4. This N3 reasoner runs in a Prolog virtual machine.

The DIVIDE server module is an executable that starts up DIVIDE. It exposes a REST API that allows to add, delete and request information about DIVIDE queries and DIVIDE components in the DIVIDE system.

3.7.2 Configuration of DIVIDE

The configuration of DIVIDE is provided through a main JSON file. It includes details about different aspects of DIVIDE. In addition, the DIVIDE components can be defined in a separate file. An example of the JSON configuration of the DIVIDE system is provided in Addendum 3.B.

Knowledge base The type of the knowledge base (e.g., Apache Jena, RDFox) should be configured, *if* it is deployed by the DIVIDE server. This is the preferred option when deploying new systems. If DIVIDE is deployed in an existing system, an existing Knowledge Base can also be used. In that case, the system will be responsible for monitoring context updates relevant to components registered to DIVIDE, and triggering the query derivation in the DIVIDE engine for those components whenever such a context update occurs.

Ontology To configure the ontology used by DIVIDE, the relevant ontology files should be specified.

Reasoner and engine The configuration of the DIVIDE reasoner and engine consists of a series of flags that allow to change the default DIVIDE behavior. For example, DIVIDE can be configured to handle TBox definitions in context graphs during the query derivation. Moreover, the parser can be configured to automatically create a variable mapping between the stream and final query based on equal variable names.

DIVIDE queries For every DIVIDE query, a separate JSON file should be linked in the configuration. This file can include the items of the internal representation of a DIVIDE query, or the end user definition of a DIVIDE query. In the latter case, the implementation of the DIVIDE query parser ensures that the parsed DIVIDE queries result in valid RSP-QL queries after the query derivation. This is achieved by validating the inputs, renaming the query variables to avoid any mismatches, ordering the input variables and static window parameters to obtain a deterministic substitution, and handling query variables in special constructs such as GROUP BY clauses.

The JSON configuration of the DIVIDE query for the running use case example can be found in Addendum 3.B.

Server For the DIVIDE server, the host and port of the exposed REST API is defined. If DIVIDE deploys the Knowledge Base as well, the port of the Knowledge Base REST API available for context updates is also specified.

DIVIDE components The components known by DIVIDE should be defined in an additional CSV file, which contains one entry per component. The properties of every component entry are separated by a semicolon.

3.7.3 Implementation of the ontology preprocessing

During the initialization of DIVIDE, the configured ontology is preprocessed with the EYE reasoner in three steps. First, an N3 copy of the full ontology is created. Second, specialized ontology-specific rules are created from the original rules taken from the OWL 2 RL profile description [10]. Starting the EYE reasoning process from these rules will reduce the computational complexity of the reasoning [73]. Third, an image of the EYE reasoner, which has already loaded the ontology and specialized rules, is compiled within Prolog. This precompiled Prolog image is the result of the ontology preprocessing. By starting the semantic reasoning step of the query derivation process from this image, the triples and rules do not need to be loaded into the EYE reasoner each time it is called during the DIVIDE query derivation. This allows to make the semantic reasoning step significantly more efficient.

Although considered infrequent, ontology changes can be handled by DIVIDE. If DIVIDE is hosting the Knowledge Base, ontology changes can be made by using the Knowledge Base REST API. Any TBox change will result in DIVIDE reloading the ontology, redoing the ontology preprocessing, and triggering the query derivation for all DIVIDE queries and components. This is a computationally intensive operation.

3.7.4 Implementation of the DIVIDE query derivation

The DIVIDE query derivation is managed by the DIVIDE engine. To decouple the scheduler of query derivation tasks from their actual parallel execution, the DIVIDE

engine manages a blocking task queue and a dedicated processing thread for every DIVIDE component in the system.

Different tasks can be scheduled by the DIVIDE engine in the blocking task queue of a DIVIDE component. The main task type is a query derivation for one or all DIVIDE queries. In case of a context change, the query derivation is scheduled for all DIVIDE queries. However, when a new DIVIDE query is added to the engine via the server API, the query derivation is only scheduled for the new DIVIDE query. In case the query execution should be performed for multiple DIVIDE queries, the query derivation steps are executed in parallel threads for every DIVIDE query. Another task type is the removal of a DIVIDE query from a component, which requires all related RSP queries to be unregistered from the component's RSP engine. This task is scheduled for all DIVIDE queries of a component when that component is removed, or for all components and one DIVIDE query when this DIVIDE query is deleted.

The following paragraphs present some further implementation details of some DIVIDE query derivation steps.

Context enrichment This step involves the execution of SPARQL queries prior to the actual query derivation. Hence, this is the only semantic step of the query derivation that is not necessarily performed by the EYE reasoner. This is the case if the queries contain SPARQL constructs that cannot be translated to a valid N3 rule. In this case, the queries are executed in Java by using Apache Jena. In the other case, the queries are translated to N3 rules which are then applied on the set consisting of triples and, if reasoning is enabled, also consisting of the ontology rules.

RSP engine query update This final step of the query derivation is not performed with the EYE reasoner. To update the query registrations at the RSP engines, the REST APIs of the RSP engine servers are used.

3.8 Evaluation set-ups

This section presents the set-up of two evaluations of the DIVIDE system. First, the performance of DIVIDE is evaluated by measuring the duration of the different key actions taken by DIVIDE during its initialization and query derivation. Second, the real-time execution of RSP-QL queries generated by the DIVIDE query derivation is evaluated. This is done by comparing the real-time DIVIDE set-up with other well-known real-time approaches.

General information about the collected data, the ontology and context, and activity rules used for these evaluations are presented in Section 3.8.1. The detailed set-ups of both individual evaluations are further described in Section 3.8.2 and

Section 3.8.3, respectively. Supportive information relevant to the evaluation set-ups of this chapter is publicly available at https://github.com/IBCNServices/DIVIDE/tree/master/swj2022.

3.8.1 Evaluation scenarios

All evaluations are performed on the eHealth use case described in Section 3.3.1. This section zooms in on the details of the evaluation scenarios of this use case.

3.8.1.1 Ontology

The ontology of the evaluation system is the Activity Recognition ontology as an extension of the existing DAHCC ontology [66], as presented in Section 3.3.2. This includes the `KBActivityRecognition` ontology and its imports. The imported ontologies include the `ActivityRecognition`, `MonitoredPerson`, `SensorsAndActuators` and `SensorsAndWearables` modules of the DAHCC ontology and its imported ontologies.

3.8.1.2 Realistic dataset for rule extraction and simulation

To properly perform the evaluations presented in this chapter, a realistic data set is used that is the result of a large scale data collection process. This data collection took place in the imec-UGent HomeLab from June 2021 until October 2021. The HomeLab is an actual standalone house located on the UGent Campus Zwijnaarde, offering a unique residential test environment for IoT services, as it is equipped with all kinds of sensors and actuators. It contains different rooms that represent a typical home: an entry hallway, a ground floor toilet, a living room and kitchen, a staircase to the first floor, and a bathroom, master bedroom, hallway and toilet on the first floor. Prior to the data collection period, a literature study, observational studies and interviews with caregivers were performed to derive the activity types that are important to detect in a patient's home. Based on these activities, a list of properties was derived that could be of relevance to observe in order to detect these activities. These properties were then translated to the required sensors, which were all installed in the HomeLab. The data collected during the data collection period in the HomeLab is used for the evaluation in two ways: to extract realistic rules for activity recognition, and to create a realistic data set for simulation during the real-time evaluations.

Throughout the data collection, data was obtained from two sources relevant to this evaluation: a wearable device, and the in-home contextual sensors. For the former, the patient was equipped with an Empatica E4 wearable device [74]. It has a 3-axis accelerometer (32 Hz) as well as different sensors to measure a person's physiological data: blood volume pulse (64 Hz) and derived inter beat interval of heart rate, galvanic skin response (4 Hz) and skin temperature (4 Hz). For the latter, as explained,

a wide range of sensors was installed in the HomeLab. These sensors measure localization, the number of people in a room, relative humidity, indoor temperature, motion, light intensity, sound, air quality, usage of water, electric power consumption of multiple devices, interaction with light switches and other buttons, the state of windows, doors, blinds, and others. During the data collection, participants labeled their activities, which were mapped by the researchers to the activities in the DAHCC ontology.

3.8.1.3 Context

The context for the evaluations, as considered by DIVIDE, consists of three main parts. The first part is the description of a patient living in a smart home, including the patient's wearables and a routine. For this part, the exact definitions in Listing 3.10 of Addendum 3.A are used. The second part is a single triple representing the patient's location in the home, which is normally derived by a specific query. For the evaluation scenarios, the location of the patient in the home will always be the bathroom. The third part is the description of all sensors, actuators and wearables of the patient's smart home with the DAHCC ontology concepts. The smart home used in this evaluation scenario is the HomeLab. The instantiated example modules `_Homelab` and `_HomelabWearables` of the DAHCC ontology contain an actual representation of all sensors, actuators and wearables used within the HomeLab. The ABox definitions in these ontology modules represent the second part of the context used for the evaluations. Note that for these evaluations, the small set of TBox definitions present in both modules are also considered part of the ontology.

3.8.1.4 Activity rules

From the data collected during the large scale data collection in the HomeLab, data-driven rule mining algorithms were created that have extracted some realistic rules that can recognize some of the DAHCC activities from the data. For the evaluations of DIVIDE in this chapter, rules for three bathroom activities are considered: toileting, showering and brushing teeth. Based on the analysis, the following rules were extracted:

- Toileting: the person present in the HomeLab is going to the toilet if a sensor that analyzes the energy consumption of the water pump has a value higher than 0.

- Showering: the person present in the HomeLab bathroom is showering if the relative humidity in the bathroom is at least 57%.

- Brushing teeth: the person present in the HomeLab bathroom is performing the brushing teeth activity if in the same time window (a) the sensor that analyzes the water running in the bathroom sink measures water running, and (b)

the activity index value of the person's acceleration (measured by a wearable) is higher than 30. The activity index based on acceleration is defined as the mean variance of the acceleration over the three axes [75].

These three activity rules have been semantically described using the Activity Recognition ontology. The resulting descriptions are part of the `KBActivityRecognition` ontology module. They are detailed in Addendum 3.C.

The three given activity rules are the only rules present in the `KBActivityRecognition` ontology module during the evaluations. To represent each activity rule within DIVIDE, a DIVIDE query is created for each rule. Because of the completely similar definition, the generic DIVIDE query corresponding to the toileting and showering activity rules is the same. The DIVIDE queries are defined as an ordered collection of SPARQL queries.

3.8.2 Performance evaluation of DIVIDE

To derive the contextually relevant RSP queries, DIVIDE performs multiple steps, both during initialization and the query derivation. To evaluate the performance of DIVIDE, the duration of the main steps is measured for the given evaluation scenarios. In concrete, the duration of the following steps is measured:

1. the ontology preprocessing step with EYE of the Activity Recognition ontology;

2. the DIVIDE query parsing step with Java of the toileting DIVIDE query defined as SPARQL input;

3. the DIVIDE query derivation step for the toileting and brushing teeth DIVIDE queries separately, split up between the different EYE steps: semantic reasoning, query extraction, input variable substitution and window parameter substitution (note that the showering DIVIDE query is the same as the toileting DIVIDE query, and is therefore not evaluated twice).

The duration of these steps is measured from within the execution of the DIVIDE server Java JAR, which is configured with the scenario's ontology and DIVIDE queries to allow performing evaluation 1 and 2. To perform evaluation 3, the full context of the scenario is sent as new context to the DIVIDE Knowledge Base server.

While evaluating the performance of the DIVIDE query parser, the correctness of the parsing is also validated.

Technical specifications The evaluation is performed on a device with a 2800 MHz quad-core Intel Core i5-7440HQ CPU and 16 GB DDR4-2400 RAM.

3.8.3 Real-time evaluation of derived DIVIDE queries

The task of DIVIDE is to manage the RSP queries on the registered RSP engines. These RSP queries are characterized by the fact that they do not require any more reasoning during their continuous evaluation, since semantic reasoning during the query derivation ensures that they are contextually relevant at every point in time. This section compares the real-time performance of evaluating these derived RSP queries on the C-SPARQL RSP engine [17].

3.8.3.1 Evaluation of DIVIDE in comparison with real-time reasoning approaches

This evaluation compares the DIVIDE real-time approach with other traditional approaches that do require real-time reasoning. The goals of the evaluation are to understand how DIVIDE compares to these traditional set-ups in terms of processing performance, and to understand the differences, advantages and drawbacks of the approaches.

To have a fair comparison, the real-time reasoning approaches should all reason within the same reasoning profile as DIVIDE, i.e., OWL 2 RL. Most approaches use RDFox [11], as this is known as one of the fastest OWL 2 RL (Datalog) reasoning engines that exist in the current state-of-the-art. Others use the Apache Jena rule reasoner.

Set-ups Different set-ups are considered for this evaluation. Most of the set-ups are streaming set-ups, meaning that they operate on windows taken from data streams. For every streaming set-up, Esper is the technology used to manage the windowing and to generate the window triggers [76].

1. DIVIDE approach using C-SPARQL without reasoning: regular C-SPARQL engine [17]. No ontology or context data is loaded into the engine, and no reasoning is performed during the continuous query evaluation.

2. Streaming RDFox: streaming version of RDFox. Consists of one engine that pipes Esper for windowing with RDFox for reasoning, via a processing queue. Initially, the ontology and context data are loaded into the data store of the RDFox engine, and a reasoning step is performed. Upon every window trigger generated by Esper, the window content is added as one event to a processing queue. When available, RDFox takes an event from the queue, incrementally adds it to the RDFox data store (i.e., it performs incremental reasoning with the event scheduled for addition), and executes the registered queries in order. If there are multiple queries registered, query X incrementally adds its results to the data store, before query $X + 1$ is executed. Finally, RDFox performs incremental reasoning with the event and all previous query outputs scheduled for deletion (i.e., incremental deletion).

3. C-SPARQL piped with (non-streaming) RDFox: Initially, the RDFox data store contains the ontology and context data, and a reasoning step is performed. The queries registered on C-SPARQL listen to the observation stream, and run continuously on the stream window data and on the ontology and context triples. The axioms in the ontology are converted to a set of rules. Rule reasoning is performed during each query evaluation using these rules by C-SPARQL, which uses the Apache Jena rule reasoner with a hybrid forward and backward reasoning algorithm. C-SPARQL sends each query result to the event stream of a regular non-streaming RDFox engine, which adds it to a processing queue. Upon processing time, it incrementally adds the event to the data store, executes the registered queries in order, and incrementally deletes the event from the data store.

4. RDFox (non-streaming): RDFox engine wrapped into a server, that listens to the observation stream. Each incoming observation is added to a queue, which is processed by a separate thread. This thread takes an event from the queue, adds it to RDFox, performs incremental reasoning, and executes the registered queries in order. If there are multiple queries registered, query X incrementally adds its results to the data store, before query $X+1$ is executed. Because this is a non-streaming version of RDFox, the event triples and triples constructed by the intermediate queries are not removed from the data store after processing.

5. Adapted Streaming RDFox: adapted streaming version of RDFox. This set-up only differs in one aspect from the original streaming RDFox set-up (2): before an event is added to RDFox, it checks the overlap between the event triples and existing triples in the data store. If overlapping triples are found, they are not added again to RDFox, and – most importantly – they are also not removed afterwards, so that no previously existing triples are removed from the data store after the event processing.

6. Semi-Streaming RDFox: mix between streaming RDFox set-up (2) and non-streaming RDFox set-up (4). This set-up only differs in one aspect from the original streaming RDFox set-up: the event triples and triples constructed by intermediate queries are not removed from the data store after processing. Hence, the only difference with the non-streaming RDFox set-up is that events are not added directly to the queue from the observation stream, but grouped together on Esper window triggers.

7. Streaming Jena: streaming version of the Apache Jena rule reasoner, similar to the streaming RDFox set-up (2). The only difference is the fact that during initialization, a set of rules is extracted from the ontology and loaded together with the ontology triples into the Apache Jena rule reasoner. Processing of events from the processing queue is done by this reasoner: it takes events,

add them to the reasoner's data model, performs forward rule reasoning using the RETE algorithm, and executes the registered queries in order. Temporal query results are also added to the reasoner's data model, which are removed after processing of the event together with the event triples, followed by a final reasoning step. This set-up uses Apache Jena v3.7.

Each set-up is deployed with an associated WebSocket server to which an external component can connect to send data to the registered data streams. Each set-up involving RDFox uses RDFox v5.2.1, via the JRDFox Java JAR, which is the Java bridge to the native RDFox engine. The RDFox data store used is the default `par-complex-nn` store, indicating a parallel data store using a complex indexing scheme with 32-bit integers.

Simulated data To create a simulation dataset to use in the evaluations, an anonymous representative portion is extracted from the dataset obtained with the large-scale data collection in the HomeLab. It contains real sensor observations of all HomeLab sensors and an Empatica E4 wearable worn by a real person living in the HomeLab for a day. Hence, the frequencies and values of the different observations are representative for a real smart home.

The simulated data for the scenarios is changed in two aspects: (i) timestamps are shifted to real-time timestamps, and (ii) the values for the sensors relevant to the evaluated activity are modified to ensure that its conditions are fulfilled all the time. In other words, the simulation for the brushing teeth scenario described below will lead to a detected brushing teeth activity during the full course of the scenario, and similarly for the other activities.

One hour of data from the anonymous data set used in this evaluation contains data of 231 different sensors, together producing 670,118 observations in this hour. 605,090 of these observations are produced by the 4 sensors of the Empatica E4 wearable, the remaining 65,028 observations are produced by 227 sensors in the HomeLab.

Specific scenarios Three specific scenarios, one for each activity rule in the general scenario, are constructed for this evaluation:

- Toileting scenario: Simulated HomeLab data for a period of 30 minutes is replayed at real rate, in batches of 1 second. For the streaming set-ups, the streaming queries are evaluated on a sliding window of 60 seconds with a sliding step of 10 seconds.

- Showering scenario: Simulated HomeLab data for a period of 20 minutes is replayed at real rate, in batches of 1 second. For the streaming set-ups, the streaming queries are evaluated on a sliding window of 60 seconds with a a sliding step of 10 seconds.

- Brushing teeth scenario: Simulated HomeLab data for a period of 30 minutes is replayed at real rate, in batches of 1 second. For the streaming set-ups, the streaming queries are evaluated on a sliding window of 30 seconds with a sliding step of 10 seconds.

The purpose of the evaluations is to measure and study the executions of the query evaluations and associated operations of the reasoner or engine, such as the semantic reasoning, both individually and progressively over time.

Replaying the data is performed by a data simulation component running on an external device in the same local network, to realistically represent the different sensor gateways. During simulation, this component connects as a client to the WebSocker server of the evaluated set-up, and sends the observations in each batch as a single message over the WebSocket connection to the appropriate data streams. This implies that one incoming event is received by the set-ups every second. Hence, the streaming set-ups will add such an event to Esper for windowing every second, while the non-streaming set-up will trigger the in-order evaluation of the registered set-ups every second.

Evaluation queries To properly compare the different set-ups for each specific scenario, different versions of the SPARQL and C-SPARQL queries are created. In concrete, the following adaptations are made:

- For set-up 1, the C-SPARQL query as outputted by DIVIDE is registered.
- For set-ups 2, 4, 5, 6 and 7, the SPARQL definition of the DIVIDE query is modified to obtain two queries registered to the single reasoning service. The first reasoning query is the stream query of the SPARQL definition, from which the graph specifications are removed. The second query is the final query of this definition. Hence, the evaluated queries that are executed with RDFox or Apache Jena are regular SPARQL queries that involve semantic reasoning and are not rewritten by DIVIDE.
- For set-up 3, the SPARQL definition of the DIVIDE query is modified to obtain two queries. The first reasoning query is derived from the stream query of the SPARQL definition: the graph specifications are removed, and the query is translated to C-SPARQL syntax by adding the relevant FROM clauses that specify the query input: the static resources and the data stream window definition. This query is registered to the C-SPARQL engine. The second query is identical to set-up 2 and is registered to RDFox.

During an evaluation run, only the quer(y)(ies) related to the activity rule of the scenario are deployed on the engines. Queries related to other activity rules or aspects like location monitoring are not registered to the engines.

Measurements For each presented set-up, the *total execution time* metric is measured for each event. This metric is defined as the time starting from a *generated event* until the timestamp where an instance of the `RoutineActivityPrediction` is returned as output by the corresponding query. In a set-up with multiple queries that are executed in order, this is always the output of the final query in the chain. The definition of a *generated event* differs for each set-up: in the streaming set-ups, this is the time of an Esper window trigger; in the non-streaming set-up 4, this is the time of an incoming set of sensor observations.

Technical specifications All evaluations are run on a typical processing device in the IoT world: an Intel NUC, model D54250WYKH. It has a 1300 MHz dual-core Intel Core i5-4250U CPU (turbo frequency 2600 MHz) and 8 GB DDR3-1600 RAM.

3.8.3.2 Real-time evaluation of derived DIVIDE queries on a Raspberry Pi

DIVIDE is considered as a semantic component in a cascading reasoning set-up in an IoT network, which involves running RSP queries on local devices. These devices can be low-end devices with few resources in an IoT context. Hence, it is important to evaluate the real-time performance of continuously executing the RSP queries outputted by DIVIDE on a low-end device like a Raspberry Pi. This is the topic of the final evaluation.

For this evaluation, only the C-SPARQL baseline set-up (1) of the previous section is considered. All other properties of this evaluation are identical to those used for the evaluation in the previous section.

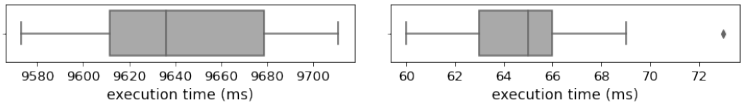
Technical specifications This evaluation is performed on a Raspberry Pi 3, Model B. This Raspberry Pi model has a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM and MicroSD storage.

3.9 Evaluation results

This section presents the results of the three evaluations described in Section 3.8. All results contain data of multiple evaluation runs, always excluding 3 warm-up and 2 cool-down runs.

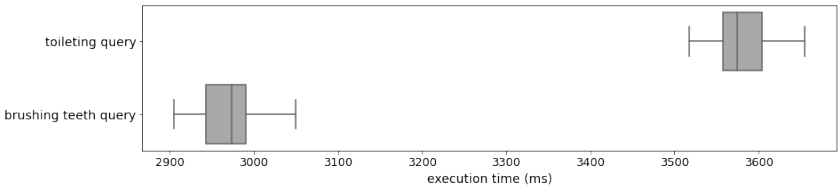
3.9.1 Performance evaluation of DIVIDE

Figure 3.3 shows the distribution of the duration of two initialization steps of the DIVIDE system: the preprocessing of the Activity Recognition ontology, and the parsing of the toileting query specified as SPARQL input. The preprocessing of the ontology on average takes 9,640 ms, with a standard deviation (SD) of only 42 ms. The average duration of the query parsing is only 64.87 ms (SD 2.76 ms). It was

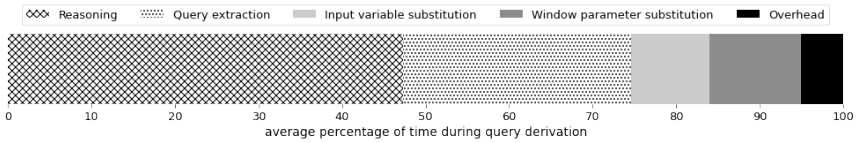


(a) Ontology preprocessing **(b)** Query parsing for toileting query

Figure 3.3: Performance results of the initialization of the DIVIDE system: boxplot distributions of total execution times per step



(a) Boxplot distribution of query derivation time for the toileting and brushing teeth DIVIDE query



(b) Relative times for query derivation substeps with EYE reasoner, averaged over all runs of the three DIVIDE queries

Figure 3.4: Performance results of the query derivation of the DIVIDE system

also validated that the parsing of the end user definition of the DIVIDE query to its internal representation was done correctly.

Figure 3.4 shows the performance results of the query derivation with DIVIDE, for the DIVIDE query corresponding to the toileting activity rule (also corresponds to the showering rule) and the DIVIDE query corresponding to the brushing teeth activity rule. Subfigure 3.4(a) shows the distribution of the duration of the query derivation for each individual query. The average durations of the query derivation are 3,578 ms (SD 38 ms) and 2,968 ms (SD 37 ms) for the toileting and brushing teeth DIVIDE queries, respectively.

Subfigure 3.4(b) shows the percentage of time taken up by the different substeps, averaged over all runs for the three DIVIDE queries. These substeps include all steps performed with the EYE reasoner: the reasoning (47.27% on average), the query extraction (27.32% on average), the input variable substitution (9.44% on average) and the window parameter substitution (10.93% on average). The remaining time (5.04% on average) is overhead of the DIVIDE implementation, including internal threading and memory operations.

3.9.2 Evaluation of DIVIDE in comparison with real-time reasoning approaches

Figure 3.5 shows the results of the comparison of the real-time evaluation with DIVIDE on a C-SPARQL engine with different real-time reasoning approaches, for the toileting query. The results show the evolution over time of the total execution time from the event generation until the routine activity prediction is generated by the engine. The measurements included in the graphs are averaged over the evaluation runs. For three setups, there are no measurements shown for the full time course of the evaluation, which takes 30 minutes. These set-ups are the pipe of C-SPARQL with RDFox set-up (3), the adapted streaming RDFox set-up (5) and the streaming Jena set-up (7). These missing measurements are caused by the systems running out of memory, causing them to stop evaluating the queries for the remainder of the scenario. The DIVIDE baseline set-up (1) has the lowest average total execution time from 960 seconds into the evaluation. Before this timestamp, the non-streaming RDFox set-up (4) is the quickest.

Figure 3.6 shows similar results for the real-time evaluation of the showering query. The same three set-ups run out of memory at a certain point, causing missing measurements for the remainder of the evaluation runs. Already after 550 seconds into the evaluation, the DIVIDE baseline set-up (1) has the lowest average total execution time.

Figure 3.7 shows similar results of the comparison of the real-time evaluation of DIVIDE with the real-time reasoning approaches, but for the brushing teeth query. The properties of the graph are similar to those of the graph presenting the results for the toileting query. In these results, only the non-streaming RDFox set-up (4) has no measurements for the full time course of the evaluation scenario due to the engine running out of memory.

In Addendum 3.D, additional results of the evaluation runs over time are included. These results visualize the distribution of the total execution times for the different set-ups at different times during the evaluation runs.

3.9.3 Real-time evaluation of derived DIVIDE queries on a Raspberry Pi

Figure 3.8 shows the results of the evaluation of the DIVIDE set-up on the Raspberry Pi 3. These results visualize the distribution of the individual execution times of the RSP queries generated by DIVIDE with the C-SPARQL baseline set-up, for the toileting, showering and brushing teeth scenarios. For the toileting query, the average total execution time is 3,666 ms (SD 318 ms). This average number is 3,699 ms (SD 286 ms) and 3,001 ms (SD 174 ms) for the showering and brushing teeth scenarios, respectively.

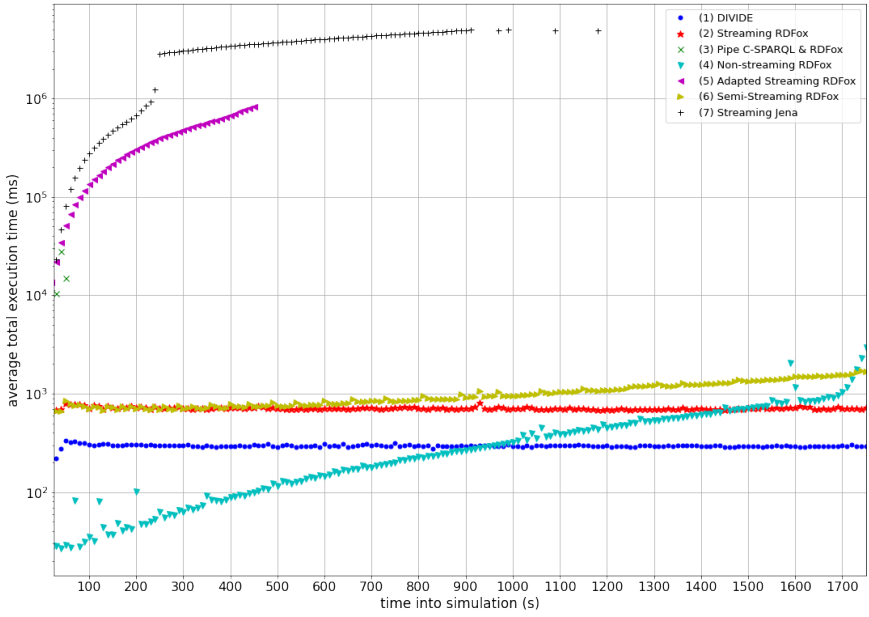


Figure 3.5: Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the toileting query. For each evaluation set-up, the results show the evolution over time of the total execution time from the generated event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the final query. For all set-ups, measurements are shown for the processed event, either incoming or windowed, at every 10 seconds. All plotted execution times are averaged over the evaluation runs.

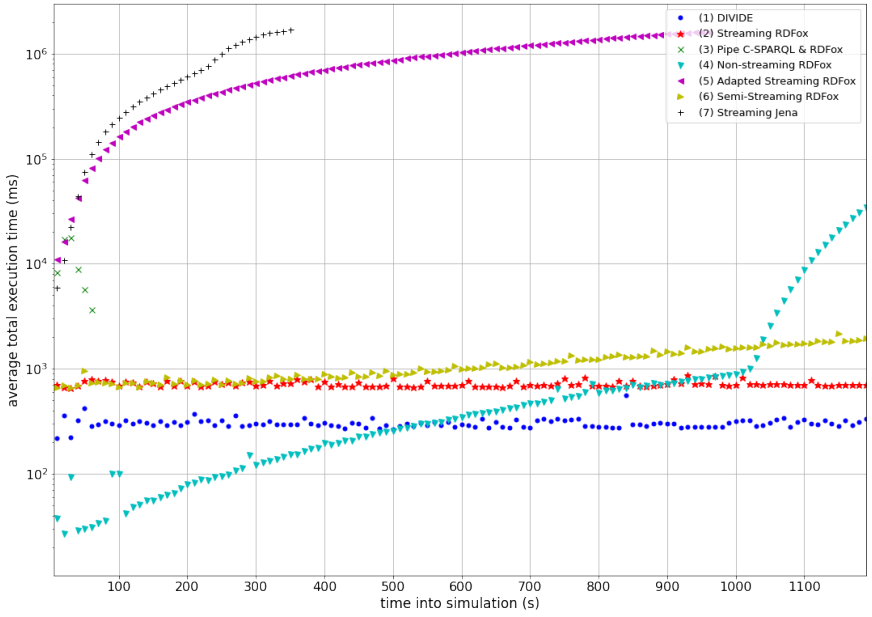


Figure 3.6: Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the showering query. For each evaluation set-up, the results show the evolution over time of the total execution time from the generated event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the final query. For all set-ups, measurements are shown for the processed event, either incoming or windowed, at every 10 seconds. All plotted execution times are averaged over the evaluation runs.

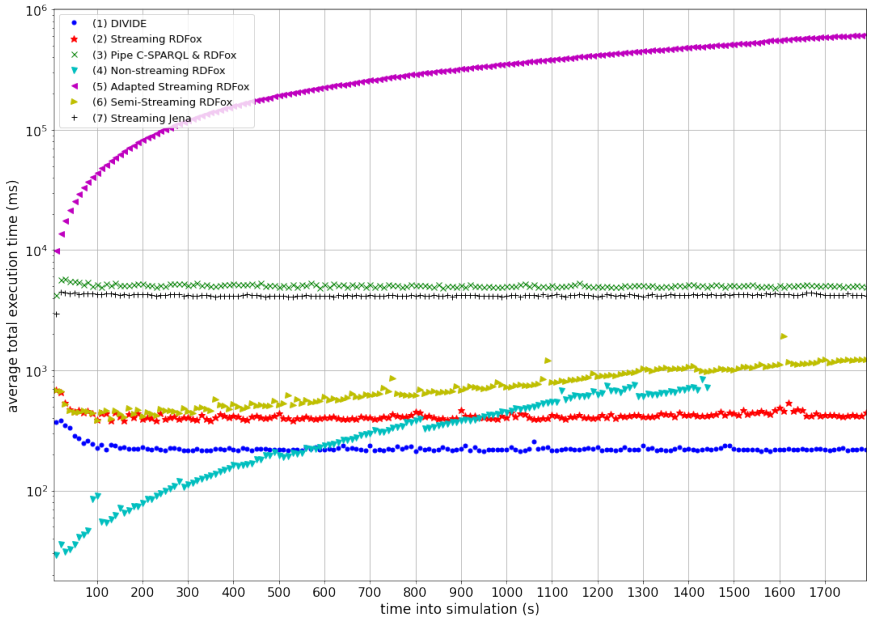


Figure 3.7: Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the brushing teeth query. For each evaluation set-up, the results show the evolution over time of the total execution time from the generated event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the final query. For all set-ups, measurements are shown for the processed event, either incoming or windowed, at every 10 seconds. All plotted execution times are averaged over the evaluation runs.

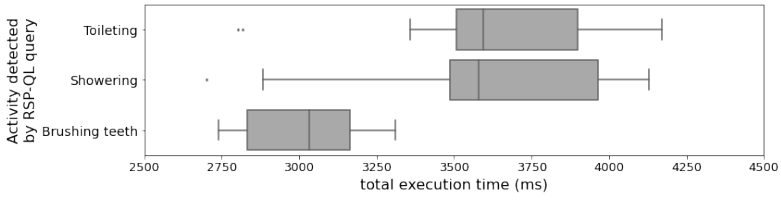


Figure 3.8: Results of evaluating the DIVIDE real-time query evaluation approach with the C-SPARQL baseline set-up (1), on a Raspberry Pi 3, Model B. The results show the total execution time distribution over the engine's runtime and multiple runs, for the toileting, showering and brushing teeth DIVIDE queries.

3.10 Discussion

Including DIVIDE as a component in a semantic IoT platform allows to perform context-aware monitoring of patients in homecare scenarios. This is possible because DIVIDE is designed to fit in a cascading architecture: it derives and manages contextually relevant RSP queries that require no additional reasoning while they are being executed, which makes them perfectly suitable to run on local low-end devices in the patient's home environment.

An end user of DIVIDE will design the IoT platform architecture for a specific use case within a certain application domain. By employing DIVIDE in a cascading reasoning architecture, DIVIDE enables privacy by design to a certain extent. As such, DIVIDE helps the end user to integrate privacy by design into the application, by following some of the privacy by design principles. More specifically, DIVIDE leaves its end users in full control to specifically define which privacy-sensitive data is exposed to the outside world. This data will typically consist of different levels of abstractions of the raw data observed by the IoT sensors. The end user control of exposed data directly follows from the definition of the DIVIDE queries: these queries exactly define which semantic concepts will be filtered by the local RSP engines, and sent over the IoT network to the central reasoner on a central server. Only the outputs of those queries will ever leave the local environment of the patient; all other data will be kept locally. This way, DIVIDE helps its end users to adhere to the embedded, user-centric, and visibility and transparency principles of privacy by design. By embedding DIVIDE into a cascading architecture, the design can consider the interests of the patients and be transparent about the data being sent over the network (i.e., the outputs of the generic RSP queries in the DIVIDE query definitions). Nevertheless, the research, implementation and integration of additional privacy solutions into the application design is required to optimally achieve privacy preservation. For example, in the described use case scenario, the patient's in-home location and detected activities comprise the only information that is ever leaving the home. While this ensures all other privacy-sensitive data is kept locally, it does not guarantee the preservation of this small set of privacy-sensitive data that is leaving the patient's environment. Such additional privacy solutions that need to be built into the application design will often be use case specific, according to the use case requirements. Hence, this requires additional, use case specific privacy research that is considered out of scope of the presented research.

With respect to security, note that the integration of DIVIDE into a semantic IoT platform does not guarantee any additional security to the system. Currently, the communication within DIVIDE is only protected via the standard SSL/TLS encryption associated with the HTTPS protocol, which is not sufficient to ensure maximum security. Hence, an additional security system or framework should be integrated into a semantic IoT platform architecture that involves DIVIDE. Existing security systems

and frameworks should be researched to achieve this. However, this is considered to be out of scope of the presented research.

A DIVIDE query is generic by nature, which ensures that you should not define one DIVIDE query for every individual reasoning or filtering task that should be performed in the use case. In the activity recognition use case scenario discussed in this chapter, one should only define a generic DIVIDE query per *type* of activity rule, instead of per activity rule individually. The generic nature of a DIVIDE query ensures that DIVIDE can derive the instantiated queries from it that are contextually relevant at any given point in time. This is achieved by listening to context updates in the knowledge base, and automatically triggering the query derivation upon a context change for all components that are affected by this context change. This is an improvement compared to systems where the management of the queries on the stream processing components of the IoT platform is still a manual, labor-intensive and thus highly impractical task. On the other hand, generic semantic queries can also be processed by reasoning engines, but while this is certainly feasible with current existing semantic reasoners for a single home environment, it might become more complex if this needs to be managed for a full network with for example many smart homes.

By deploying DIVIDE in a cascading architecture, more benefits are obtained than solely the privacy control for the end user, generic query definition and context-awareness. Since the high frequency and high volume data streams are processed locally, this data should not be transferred over the network. This significantly reduces network bandwidth usage and network delay impacting the system's performance. In addition, the data does not need to be processed by the central reasoner, which now only receives the outputs of the local RSP queries to do further processing. As such, the main resources of the server can be saved for the high-priority situations. In the presented use case scenario, an example is when an activity is detected that is not in the patient's routine: when this prediction is received by the central reasoner, it can investigate the cause of the issue and trigger further actions such as generating an alarm when needed. Meanwhile, the server resources can also be used by DIVIDE to derive the updated location monitoring query to ensure that the patient's location is followed up more closely.

When DIVIDE is used as a component in a semantic IoT platform to derive and manage the local RSP queries, it is of course important that the queries derived by DIVIDE have a good performance that is comparable to existing state-of-the-art stream reasoning systems. The results of this comparison with the C-SPARQL RSP engine running on an Intel NUC device demonstrate that the filtering RSP queries perform very well for the different activity detection queries that each correspond to a generic real-time reasoning set-up. The results show how the C-SPARQL queries are only outperformed by the classic non-streaming RDFox reasoning engine if you only look at the processing of single events. This can easily be explained by the fact that the events processed by this RDFox set-up contain fewer observations, and thus triples,

than the events processed by C-SPARQL, which are larger batches of data grouped in data windows. Hence, due to the incremental reasoning in RDFox, this set-up initially performs best. However, looking at the evolution of the total execution times over time, the DIVIDE baseline set-up starts to perform better after a while. This is because the performance of the DIVIDE set-up stays constant over time, while the total execution time of the queries on the RDFox set-up increases over time because events are not removed from the data store, increasing the size of the data store on every execution. Therefore, we have also included a comparison with a streaming version of RDFox. This set-up also performs constant over time, and is outperformed by a slight margin only by DIVIDE. This is mainly because RDFox still has to do some reasoning, which, even though this happens very efficiently with RDFox, is not required for the evaluation of the RSP queries with C-SPARQL in the baseline set-up. The streaming set-up of RDFox used in the evaluations makes a few assumptions that can still be optimized by looking at overlapping events and ensuring they are not removed after the processing of an event. However, in this optimized, adapted streaming RDFox setup, the processing of incoming events cannot keep up with the rate of the windowed events, causing the processing delay to build up. This leads to very long query execution times and memory issues in some cases. Moreover, looking at the results that involve reasoning with Apache Jena, it is clear that the set-ups using this semantic reasoner perform way worse than the DIVIDE and optimal RDFox set-ups. This is also true for the pipe of C-SPARQL with RDFox, in which C-SPARQL is performing rule-based reasoning with Apache Jena in the first query. This reasoning step causes the bad performance entirely on its own. This learns that using the built-in rule reasoning support of C-SPARQL is not efficient compared to alternative set-ups. As a conclusion, over time, DIVIDE performs comparable or even slightly better than the best RDFox set-ups, making it an ideal solution to integrate in a semantic platform to manage the local RSP queries, given the other main advantages. Ideally, this is combined in the cascading architecture with a central reasoner that does use a performant semantic reasoner such as RDFox.

In IoT networks, devices with resources comparable to those of an Intel NUC are often unavailable locally. Therefore, it is important that the RSP queries can also be continuously executed on low-end devices with fewer resources. Otherwise, the data would still have to be sent to other devices with more resources running more centrally in the network that would then host the RSP engines. This would imply that all other advantages related to privacy, network usage and server resources do no longer apply. Therefore, the evaluation of the C-SPARQL baseline set-up was also performed on a Raspberry Pi. The results demonstrate that the queries can still be efficiently and consistently executed on such devices with way fewer resources than an Intel NUC. Specifically for this evaluation, the queries take approximately 10 times longer than on the Intel NUC, but take still well below the query execution frequency of 10 seconds. This is an additional advantage when deploying a system involving

DIVIDE, as no large scale investment in expensive high-end hardware is required. In real set-ups, actually deploying a Raspberry Pi may however not be very practical or realistic. However, the resources of a Raspberry Pi are very comparable to other local devices such as wearable devices like the smartwatches in Samsung Galaxy Watch or Apple Watch series. Note that RDFox can also be used instead of C-SPARQL to run the queries derived by DIVIDE on a local low-end device, since RDFox can successfully run on an ARM based edge device like a Raspberry Pi or a smartphone as well [13]. This implies that the use of a RDFox set-up would also ensure that data can be processed locally instead of being sent to a server. Also note that RDFox is able to handle any arbitrary OWL 2 RL ontology, including recursive ones.

Up to now, we have only looked at the real-time evaluation of RSP queries derived by DIVIDE. They perform well in realistic homecare monitoring environments, but another important aspect is the performance of DIVIDE itself. The results of the DIVIDE performance evaluation show that the main portion of time during the initialization of DIVIDE is taken by the preprocessing of the ontology. Of course, the duration of the preprocessing depends on the number of triples and axioms defined in the ontology, which is use case specific. In any case, this is a task that should only happen once, given the assumption in DIVIDE that ontology updates do not happen. Nevertheless, DIVIDE does support ontology updates, but they require the ontology preprocessing to be redone. Besides the initialization, it is important to inspect the duration the query derivation process when a context change is observed. For this step, the performance results show that for the given evaluation use case scenario, the query derivation typically takes around 3 to 4 seconds. This is an order of magnitude higher than the time needed to perform real-time reasoning with RDFox during the query evaluation on an incoming event. However, the execution frequency of the query derivation is a few orders of magnitude smaller than the frequency of the event processing: events are processed on every window trigger or incoming observation, which is every 10 seconds or every second in the evaluation use case scenario. As you are not expecting a context change every 10 seconds, this shows that the performance results of the query derivation step are perfectly acceptable. In addition, the results show that the largest portion of the time is taken up by the different steps performed with the EYE reasoner. The biggest portion of the time, almost 50%, is spent on generating the proof with the EYE reasoner. The results show that only less than 5% of the query derivation step is overhead induced by the DIVIDE implementation.

When integrating DIVIDE into a semantic IoT platform, it is important to note that DIVIDE considers all semantic specifications to be accurate. Hence, DIVIDE considers it the responsibility of its end users to ensure that the semantic definitions in the knowledge base and the DIVIDE queries are correctly defined. For example, in the use case scenario described in this chapter, DIVIDE assumes that all activity rules defined in the Activity Recognition ontology correctly detect the corresponding activity types. Thus, DIVIDE will not take any measures itself to avoid any misleading

of the system: if the end user wants to abuse DIVIDE to generate incorrect outputs, such as incorrectly detected activities in the described use case scenario, this is possible. Hence, it is important that all semantic definitions and DIVIDE queries of a use case are validated before they are integrated into DIVIDE.

To be able to use DIVIDE in a real IoT platform set-up, it is important that DIVIDE is practically usable. Therefore, we have implemented DIVIDE in a way that tries to maximize its practical usability. First, DIVIDE is available as an executable Java JAR component that can easily be run in a server environment, allowing for easy integration into an existing IoT platform. The main configuration of the server, engine, DIVIDE queries and components can be easily created and modified with straightforward JSON and CSV files. Importantly, DIVIDE also does not hinder RSP engines to have active queries managed manually or by other system components, ensuring that the inclusion of DIVIDE into a semantic platform is not an all-or-nothing choice. In addition, the REST API exposed by the DIVIDE server implies that the configuration of DIVIDE is not fixed: components and DIVIDE queries can be easily added or removed, increasing the flexibility of the system. The internal implementation ensures that such changes are correctly handled and reflected on the RSP engines as well. Moreover, the implementation of the query parser allows the flexible and straightforward end user definition of a DIVIDE query. This allows existing sets of queries to be used with DIVIDE to perform semantically equivalent tasks after only a small configuration effort. This way, no inner details of DIVIDE need to be known by end users who want to integrate it into their system. Our implementation of the parser also validates the DIVIDE query definitions given by the end user, and provides a human-friendly explanation about what is wrong in case the input is invalid. As a result, we believe that DIVIDE is perfectly suited in an IoT set-up where it is deployed in a cascading architecture.

3.11 Conclusion

This chapter has presented the DIVIDE system. DIVIDE is designed as a semantic component that can automatically and adaptively derive and manage the queries of the stream processing components in a semantic IoT platform, in a context-aware manner. Through a specific homecare monitoring use case, this chapter has shown how DIVIDE can divide the active queries across a cascading IoT set-up, and conquer the issues of existing systems by fulfilling important requirements related to data privacy preservation, performance, and usability.

Reaching back to the research objectives outlined in Section 3.1, we have achieved these in this chapter with DIVIDE in the following ways:

1. DIVIDE automatically triggers the derivation of the semantic queries of a stream processing component when changes are observed to context information that is relevant to that specific component. This way, DIVIDE automatically ensures that the active queries on each component are contextually relevant at all times. This process is context-aware and adaptive by design, minimizing the manual configuration effort for the end user to the initial query definition only. Once the system is deployed, no configuration changes are required anymore.

2. By performing semantic reasoning on the current context during the query derivation, DIVIDE ensures that the resulting stream processing queries can perform all relevant monitoring tasks without doing real-time reasoning. The evaluations on the use case scenario demonstrate how this ensures that DIVIDE performs comparable or even slightly better than state-of-the-art stream reasoning set-ups involving RDFox in terms of query execution times. This implies that the queries can also be executed in real-time on low-end devices with few resources, as demonstrated by the evaluations. The cascading architecture in which DIVIDE is adopted ensures minimal network congestion and optimal usage of the central resources of the network.

3. Through the definition of a DIVIDE query, an end user can make the window parameters of the stream processing queries context-dependent with DIVIDE.

4. By adopting a cascading reasoning architecture, DIVIDE manages the queries for the stream processing components that are running on local IoT devices. Integrating DIVIDE into a semantic IoT platform enables privacy by design to a certain extent: it leaves the end users, who design the platform architecture for a specific use case, in full control to specify in the DIVIDE query definitions which privacy-sensitive data is kept locally by the local stream processing queries, and which data (abstractions) in the query outputs are sent over the IoT network to the central services.

5. Generic queries in DIVIDE can be easily defined by only slightly adapting existing SPARQL or RSP-QL queries, ensuring DIVIDE is practically usable.

There are multiple interesting future pathways related to DIVIDE that are worth investigating. First, the cascading architectural set-up in which DIVIDE is ideally deployed can be further exploited. By including the monitoring of device, network and/or stream characteristics into DIVIDE, the distribution of semantic stream processing queries across the IoT network could be dynamically adapted to optimize both local and global system performance. Such a monitor could also exploit the dynamic window parameter substitution functionality of DIVIDE to adapt these parameters to the monitored conditions. Second, the current implementation of DIVIDE only

supports use cases that reason in the OWL 2 RL profile. However, the EYE reasoner used supports extending the rule set to obtain higher expressivity. Doing so would introduce support for higher expressivity use cases in DIVIDE.

Funding

This research is part of the imec.ICON project PROTEGO (HBC.2019.2812), co-funded by imec, VLAIO, Televic, Amaron, Z-Plus and ML2Grow. Bram Steenwinckel (1SA0219N) is funded by a strategic base research grant of Fund for Scientific Research Flanders (FWO), Belgium. Pieter Bonte (1266521N) is funded by a postdoctoral fellowship of FWO.

Availability of data and materials

Supportive information relevant to the evaluation set-ups of this chapter is publicly available at https://github.com/IBCNServices/DIVIDE/tree/master/swj2022. This page also refers to the source code of DIVIDE at https://github.com/IBCNServices/DIVIDE/tree/master/src/divide-central, additional details of the DAHCC ontology at https://dahcc.idlab.ugent.be, and the described dataset used to construct the evaluation rules and simulation data at https://dahcc.idlab.ugent.be/dataset.html.

References

[1] K. Jaiswal and V. Anand. *A Survey on IoT-Based Healthcare System: Potential Applications, Issues, and Challenges*. In A. A. Rizvanov, B. K. Singh, and P. Ganasala, editors, Advances in Biomedical Engineering and Technology, pages 459–471. Springer Singapore, 2021. doi:10.1007/978-981-15-6329-4_38.

[2] X. Su, J. Riekki, J. K. Nurminen, J. Nieminen, and M. Koskimies. *Adding semantics to Internet of Things*. Concurrency and Computation: Practice and Experience, 27(8):1844–1860, 2015. doi:10.1002/cpe.3203.

[3] C. C. Aggarwal, N. Ashish, and A. Sheth. *The Internet of Things: A Survey from the Data-Centric Perspective*. In C. C. Aggarwal, editor, Managing and Mining Sensor Data, pages 383–428. Springer US, 2013. doi:10.1007/978-1-4614-6309-2_12.

[4] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. *Semantics for the Internet of Things: Early Progress and Back to the Future*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012. doi:10.4018/jswis.2012010101.

[5] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. *Stream reasoning: A survey and outlook*. Data Science, 1(1-2):59–83, 2017. doi:10.3233/DS-170006.

[6] P. Bonte, F. Ongenae, and F. De Turck. *Subset reasoning for event-based systems*. IEEE Access, 7:107533–107549, 2019. doi:10.1109/ACCESS.2019.2932937.

[7] K. Abouelmehdi, A. Beni-Hssane, H. Khaloufi, and M. Saadi. *Big data security and privacy in healthcare: A Review*. Procedia Computer Science, 113:73–80, 2017. doi:10.1016/j.procs.2017.08.292.

[8] A. Cavoukian. *Privacy by design*, 2009. Accessed: 2022-09-25. Available from: https://www.ipc.on.ca/wp-content/uploads/Resources/7foundationalprinciples.pdf.

[9] M. Dürst and M. Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC – Proposed Standard 3987, Internet Engineering Task Force (IETF), 2005. Available from: https://datatracker.ietf.org/doc/rfc3987/.

[10] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. *OWL 2 Web Ontology Language Profiles (Second Edition)*. W3C Recommendation, World Wide Web Consortium (W3C), 2012. Available from: https://www.w3.org/TR/owl2-profiles/.

[11] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. *RDFox: A highly-scalable RDF store*. In The Semantic Web - ISWC 2015: Proceedings, Part II of the 14th International Semantic Web Conference, pages 3–20, Cham, Switzerland, 2015. Springer. doi:10.1007/978-3-319-25010-6_1.

[12] J. Urbani, C. Jacobs, and M. Krötzsch. *Column-oriented datalog materialization for large knowledge graphs*. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. OJS/PKP, 2016. doi:10.1609/aaai.v30i1.9993.

[13] J. Lee, T. Hwang, J. Park, Y. Lee, B. Motik, and I. Horrocks. *A context-aware recommendation system for mobile devices*. In K. Taylor, R. Goncalves, F. Lecue, and J. Yan, editors, Proceedings of the ISWC 2020 Demos and Industry Tracks: From Novel Ideas to Industrial Practice, co-located with 19th International Semantic Web Conference (ISWC 2020). CEUR Workshop Proceedings, 2020. Available from: https://ceur-ws.org/Vol-2721/paper489.pdf.

[14] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler. *N3Logic: A logical framework for the World Wide Web*. Theory and Practice of Logic Programming, 8(3):249–269, 2008. doi:10.1017/S1471068407003213.

[15] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride. *RDF 1.1 concepts and abstract syntax*. W3C Recommendation, World Wide Web Consortium (W3C), 2014. Available from: https://www.w3.org/TR/rdf11-concepts/.

[16] X. Su, E. Gilman, P. Wetz, J. Riekki, Y. Zuo, and T. Leppänen. *Stream reasoning for the Internet of Things: Challenges and gap analysis*. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), pages 1–10, New York, NY, USA, 2016. Association for Computing Machinery (ACM). doi:10.1145/2912845.2912853.

[17] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. *C-SPARQL: a continuous query language for RDF data streams*. International Journal of Semantic Computing, 4(1):3–25, 2010. doi:10.1142/S1793351X10000936.

[18] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. *A native and adaptive approach for unified processing of linked streams and linked data*. In The Semantic Web - ISWC 2011: Proceedings, Part I of the 10th International Semantic Web Conference, pages 370–388, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-25073-6_24.

[19] R. Tommasini and E. Della Valle. *Yasper 1.0: Towards an RSP-QL Engine*. In Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks, co-located with 16th International Semantic Web Conference (ISWC 2017). CEUR Workshop Proceedings, 2017. Available from: https://ceur-ws.org/Vol-1963/paper487.pdf.

[20] R. Tommasini, P. Bonte, F. Ongenae, and E. Della Valle. *RSP4J: An API for RDF Stream Processing*. In R. Verborgh, K. Hose, H. Paulheim, P.-A. Champin, M. Maleshkova, O. Corcho, P. Ristoski, and M. Alam, editors, The Semantic

Web: Proceedings of the 18th International Conference, ESWC 2021, pages 565–581, Cham, Switzerland, 2021. Springer. doi:10.1007/978-3-030-77385-4_34.

[21] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte, and O. Corcho. *RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems*. International Journal on Semantic Web and Information Systems (IJSWIS), 10(4):17–44, 2014. Available from: https://dl.acm.org/doi/10.5555/2795081.2795083.

[22] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. *Incremental reasoning on streams and rich background knowledge*. In The Semantic Web: Research and Applications: Proceedings, Part I of the 7th Extended Semantic Web Conference, ESWC 2010, pages 1–15, Cham, Switzerland, 2010. Springer. doi:10.1007/978-3-642-13486-9_1.

[23] S. Komazec, D. Cerri, and D. Fensel. *Sparkwave: continuous schema-enhanced pattern matching over RDF data streams*. In Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS 2012), pages 58–68, New York, NY, USA, 2012. Association for Computing Machinery (ACM). doi:10.1145/2335484.2335491.

[24] B. Motik, Y. Nenov, R. E. F. Piro, and I. Horrocks. *Incremental update of datalog materialisation: the backward/forward algorithm*. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence. OJS/PKP, 2015. doi:10.1609/aaai.v29i1.9409.

[25] J. Urbani, A. Margara, C. Jacobs, F. v. Harmelen, and H. Bal. *Dynamite: Parallel materialization of dynamic RDF data*. In The Semantic Web - ISWC 2013: Proceedings, Part I of the 12th International Semantic Web Conference, pages 657–672, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-41335-3_41.

[26] F. Lécué. *Diagnosing Changes in An Ontology Stream: A DL Reasoning Approach*. In Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence. OJS/PKP, 2012. doi:10.1609/aaai.v26i1.8113.

[27] E. Thomas, J. Z. Pan, and Y. Ren. *TrOWL: Tractable OWL 2 reasoning infrastructure*. In The Semantic Web: Research and Applications: Proceedings, Part II of the 7th Extended Semantic Web Conference, ESWC 2010, pages 431–435, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-13489-0_38.

[28] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth. *StreamRule: a nonmonotonic stream reasoning system for the semantic web*. In Web Reasoning and Rule Systems: Proceedings of the 7th International Conference, RR 2013, pages 247–252, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39666-3_23.

[29] H. Beck, M. Dao-Tran, and T. Eiter. *LARS: A logic-based framework for analytic reasoning over streams*. Artificial Intelligence, 261:16–70, 2018. doi:10.1016/j.artint.2018.04.003.

[30] H. R. Bazoobandi, H. Beck, and J. Urbani. *Expressive Stream Reasoning with Laser*. In The Semantic Web - ISWC 2017: Proceedings, Part I of the 16th International Semantic Web Conference, pages 87–103, Cham, Switzerland, 2017. Springer. doi:10.1007/978-3-319-68288-4_6.

[31] X. Ren, O. Curé, H. Naacke, and G. Xiao. *BigSR: real-time expressive RDF stream reasoning on modern Big Data platforms*. In 2018 IEEE International Conference on Big Data (Big Data), pages 811–820, New York, NY, USA, 2018. IEEE. doi:10.1109/BigData.2018.8621947.

[32] P. Bonte, R. Tommasini, F. De Turck, F. Ongenae, and E. D. Valle. *C-Sprite: Efficient Hierarchical Reasoning for Rapid RDF Stream Processing*. In Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, pages 103–114, 2019. doi:10.1145/3328905.3329502.

[33] T.-L. Pham, M. I. Ali, and A. Mileo. *Enhancing the scalability of expressive stream reasoning via input-driven parallelization*. Semantic Web, 10(3):457–474, 2019. doi:10.3233/SW-180330.

[34] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. *EP-SPARQL: a unified language for event processing and stream reasoning*. In Proceedings of the 20th International Conference on World Wide Web (WWW 2011), pages 635–644, New York, NY, USA, 2011. Association for Computing Machinery (ACM). doi:10.1145/1963405.1963495.

[35] D. Luckham. *The Power of Events: An introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, 2002.

[36] D. Dell'Aglio, M. Dao-Tran, J.-P. Calbimonte, D. Le Phuoc, and E. Della Valle. *A query model to capture event pattern matching in RDF stream processing query languages*. In Knowledge Engineering and Knowledge Management: Proceedings of the 20th International Conference, EKAW 2016, pages 145–162, Cham, Switzerland, 2016. Springer. doi:10.1007/978-3-319-49004-5_10.

[37] D. C. Luckham and B. Frasca. *Complex Event Processing in Distributed Systems*. Computer Systems Laboratory Technical Report CSL-TR-98-754, Stanford University, 1998. Available from: https://www.unix.com/pdf/CEP_in_distributed_systems.pdf.

[38] J.-P. Calbimonte, J. Mora, and O. Corcho. *Query rewriting in RDF stream processing*. In The Semantic Web: Latest Advances and New Domains: Proceedings of the

13th International Conference, ESWC 2016, pages 486–502, Cham, Switzerland, 2016. Springer. doi:10.1007/978-3-319-34129-3_30.

[39] D. Puiu, P. Barnaghi, R. Tönjes, D. Kümper, M. I. Ali, A. Mileo, J. X. Parreira, M. Fischer, S. Kolozali, N. Farajidavar, F. Gao, T. Iggena, T.-L. Pham, C.-S. Nechifor, D. Puschmann, and J. Fernandes. *CityPulse: Large Scale Data Analytics Framework for Smart Cities*. IEEE Access, 4:1086–1108, 2016. doi:10.1109/ACCESS.2016.2541999.

[40] F. Heintz, J. Kvarnström, and P. Doherty. *Bridging the sense-reasoning gap: DyKnow - stream-based middleware for knowledge processing*. Advanced Engineering Informatics, 24(1):14–26, 2010. doi:10.1016/j.aei.2009.08.007.

[41] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. *Stream reasoning and complex event processing in ETALIS*. Semantic Web, 3(4):397–407, 2012. doi:10.3233/SW-2011-0053.

[42] Ö. L. Özçep, R. Möller, and C. Neuenstadt. *A stream-temporal query language for ontology based data access*. In KI 2014: Advances in Artificial Intelligence: Proceedings of the 37th Annual German Conference on AI, pages 183–194, Cham, Switzerland, 2014. Springer. doi:10.1007/978-3-319-11206-0_18.

[43] G. Xiao, L. Ding, B. Cogrel, and D. Calvanese. *Virtual knowledge graphs: An overview of systems and use cases*. Data Intelligence, 1(3):201–223, 2019. doi:10.1162/dint_a_00011.

[44] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. Van Harmelen. *Towards expressive stream reasoning*. In Semantic Challenges in Sensor Networks, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. doi:10.4230/DagSemProc.10042.4.

[45] P. Bonte, R. Tommasini, E. Della Valle, F. De Turck, and F. Ongenae. *Streaming MASSIF: cascading reasoning for efficient processing of iot data streams*. Sensors, 18(11):3832, 2018. doi:10.3390/s18113832.

[46] I. Kalamaras, N. Kaklanis, K. Votis, and D. Tzovaras. *Towards Big Data Analytics in Large-Scale Federations of Semantically Heterogeneous IoT Platforms*. In L. Iliadis, I. Maglogiannis, and V. Plagianakos, editors, Artificial Intelligence Applications and Innovations: Proceedings of AIAI 2018 IFIP 12.5 International Workshops, pages 13–23, Cham, Switzerland, 2018. Springer. doi:10.1007/978-3-319-92016-0_2.

[47] P. Chamoso, A. González-Briones, F. De La Prieta, G. K. Venyagamoorthy, and J. M. Corchado. *Smart city as a distributed platform: Toward a system for citizen-oriented management*. Computer Communications, 152:323–332, 2020. doi:10.1016/j.comcom.2020.01.059.

[48] F. Cirillo, G. Solmaz, E. L. Berz, M. Bauer, B. Cheng, and E. Kovacs. *A standard-based open source IoT platform: FIWARE*. IEEE Internet of Things Magazine, 2(3):12–18, 2019. doi:10.1109/IOTM.0001.1800022.

[49] Sofia2. *Sofia2 – Technology for Innovators*, 2020. Accessed: 2022-03-10. Available from: https://sofia2.com.

[50] S. Soursos, I. P. Žarko, P. Zwickl, I. Gojmerac, G. Bianchi, and G. Carrozzo. *Towards the cross-domain interoperability of IoT platforms*. In 2016 European conference on networks and communications (EuCNC), pages 398–402, New York, NY, USA, 2016. IEEE. doi:10.1109/EuCNC.2016.7561070.

[51] A. Felfernig, S. P. Erdeniz, P. Azzoni, M. Jeran, A. Akcay, and C. Doukas. *Towards configuration technologies for IoT gateways*. In Proceedings of the 18th International Configuration Workshop, pages 73–76, 2016. Available from: https://ase.ist.tugraz.at/wp-content/uploads/sites/34/2016/07/configuration-technologies-iot-16.pdf.

[52] A. Bröring, S. Schmid, C.-K. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. Le Phuoc, J. Mitic, D. Anicic, and E. Teniente. *Enabling IoT ecosystems through platform interoperability*. IEEE Software, 34(1):54–61, 2017. doi:10.1109/MS.2017.2.

[53] A. Cimmino, V. Oravec, F. Serena, P. Kostelnik, M. Poveda-Villalón, A. Tryferidis, R. García-Castro, S. Vanya, D. Tzovaras, and C. Grimm. *VICINITY: IoT semantic interoperability based on the web of things*. In 15th International Conference on Distributed Computing in Sensor Systems (DCOSS), pages 241–247, New York, NY, USA, 2019. IEEE. doi:10.1109/DCOSS.2019.00061.

[54] M. Ganzha, M. Paprzycki, W. Pawłowski, P. Szmeja, and K. Wasielewska. *Semantic interoperability in the Internet of Things: An overview from the INTER-IoT perspective*. Journal of Network and Computer Applications, 81:111–124, 2017. doi:10.1016/j.jnca.2016.08.007.

[55] A. Javed, S. Kubler, A. Malhi, A. Nurminen, J. Robert, and K. Främling. *bIoTope: Building an IoT Open Innovation Ecosystem for Smart Cities*. IEEE Access, 8:224318–224342, 2020. doi:10.1109/ACCESS.2020.3041326.

[56] G. Marques, A. K. Bhoi, and K. Hareesha, editors. *IoT in Healthcare and Ambient Assisted Living*. Springer Singapore, 2021. doi:10.1007/978-981-15-9897-5.

[57] M. Javaid and I. H. Khan. *Internet of Things (IoT) enabled healthcare helps to take the challenges of COVID-19 Pandemic*. Journal of Oral Biology and Craniofacial Research, 11(2):209–214, 2021. doi:10.1016/j.jobcr.2021.01.015.

[58] H. K. Bharadwaj, A. Agarwal, V. Chamola, N. R. Lakkaniga, V. Hassija, M. Guizani, and B. Sikdar. *A review on the role of machine learning in enabling IoT based healthcare applications*. IEEE Access, 9:38859–38890, 2021. doi:10.1109/ACCESS.2021.3059858.

[59] R. Zgheib, S. Kristiansen, E. Conchon, T. Plageman, V. Goebel, and R. Bastide. *A scalable semantic framework for IoT healthcare applications*. Journal of Ambient Intelligence and Humanized Computing, 2020. doi:10.1007/s12652-020-02136-2.

[60] S. Jabbar, F. Ullah, S. Khalid, M. Khan, and K. Han. *Semantic interoperability in heterogeneous IoT infrastructure for healthcare*. Wireless Communications and Mobile Computing, 2017, 2017. doi:10.1155/2017/9731806.

[61] F. Ullah, M. A. Habib, M. Farhan, S. Khalid, M. Y. Durrani, and S. Jabbar. *Semantic interoperability for big-data in heterogeneous IoT infrastructure for healthcare*. Sustainable Cities and Society, 34:90–96, 2017. doi:10.1016/j.scs.2017.06.010.

[62] F. Ali, S. R. Islam, D. Kwak, P. Khan, N. Ullah, S.-j. Yoo, and K. S. Kwak. *Type-2 fuzzy ontology–aided recommendation systems for IoT–based healthcare*. Computer Communications, 119:138–155, 2018. doi:10.1016/j.comcom.2017.10.005.

[63] V. Subramaniyaswamy, G. Manogaran, R. Logesh, V. Vijayakumar, N. Chilamkurti, D. Malathi, and N. Senthilselvan. *An ontology-driven personalized food recommendation in IoT-based healthcare system*. The Journal of Supercomputing, 75(6):3184–3216, 2019. doi:10.1007/s11227-018-2331-8.

[64] P. Schaar. *Privacy by design*. Identity in the Information Society, 3(2):267–274, 2010. doi:10.1007/s12394-010-0055-x.

[65] C. Kurtz, M. Semmann, and T. Böhmann. *Privacy by design to comply with GDPR: a review on third-party data processors*. In Proceedings of the 24th Americas Conference on Information Systems (AMCIS) 2018, 2018. Available from: https://aisel.aisnet.org/amcis2018/Security/Presentations/36/.

[66] B. Steenwinckel, M. De Brouwer, M. Stojchevska, J. Van Der Donckt, J. Nelis, J. Ruyssinck, J. van der Herten, K. Casier, J. Van Ooteghem, P. Crombez, F. De Turck, S. Van Hoecke, and F. Ongenae. *Data Analytics For Health and Connected Care: Ontology, Knowledge Graph and Applications*. In Proceedings of the 16th EAI International Conference on Pervasive Computing Technologies for Healthcare (EAI PervasiveHealth 2022), 2022. Available from: https://dahcc.idlab.ugent.be.

[67] L. Daniele, F. den Hartog, and J. Roes. *Created in Close Interaction with the Industry: The Smart Appliances REFerence (SAREF) Ontology*. In Formal Ontologies Meet Industry, pages 100–112, Cham, Switzerland, 2015. Springer. doi:10.1007/978-3-319-21545-7_9.

[68] M. Girod-Genet, L. N. Ismail, M. Lefrançois, and J. Moreira. *ETSI TS 103 410-8 V1.1.1 (2020-07): SmartM2M; Extension to SAREF; Part 8: eHealth/Ageing-well Domain*. Technical report, ETSI Technical Committee Smart Machine-to-Machine communications (SmartM2M), 2020. Available from: https://www.etsi.org/deliver/etsi_ts/103400_103499/10341008/01.01.01_60/ts_10341008v010101p.pdf.

[69] I. Esnaola-Gonzalez, J. Bermúdez, I. Fernández, and A. Arnaiz. *Two Ontology Design Patterns toward Energy Efficiency in Buildings*. In Proceedings of the 9th Workshop on Ontology Design and Patterns (WOP 2018), co-located with 17th International Semantic Web Conference (ISWC 2018), pages 14–28. CEUR Workshop Proceedings, 2018. Available from: https://ceur-ws.org/Vol-2195/pattern_paper_2.pdf.

[70] M. De Brouwer, F. Ongenae, P. Bonte, and F. De Turck. *Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions*. Sensors, 18(10):3514, 2018. doi:10.3390/s18103514.

[71] D. Arndt, P. Bonte, A. Dejonghe, R. Verborgh, F. De Turck, and F. Ongenae. *SENSdesc: Connect sensor queries and context*. In 11th International Joint Conference on Biomedical Engineering Systems and Technologies, pages 1–8, 2018. doi:10.5220/0006733106710679.

[72] R. Verborgh and J. De Roo. *Drawing Conclusions from Linked Data on the Web: The EYE Reasoner*. IEEE Software, 32(3):23–27, 2015. doi:10.1109/MS.2015.63.

[73] D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle, and E. Mannens. *Improving OWL RL reasoning in N3 by using specialized rules*. In International Experiences and Directions Workshop on OWL (OWLED) 2015, pages 93–104. Springer, 2015. doi:10.1007/978-3-319-33245-1_10.

[74] Empatica. *E4 wristband*, 2020. Accessed: 2020-10-23. Available from: https://www.empatica.com/research/e4.

[75] J. Bai, C. Di, L. Xiao, K. R. Evenson, A. Z. LaCroix, C. M. Crainiceanu, and D. M. Buchner. *An activity index for raw accelerometry data and its comparison with other activity metrics*. PLoS ONE, 11(8):e0160644, 2016. doi:10.1371/journal.pone.0160644.

[76] EsperTech. *Esper*, 2022. Accessed: 2022-03-29. Available from: https://www.espertech.com/esper.

Addendum 3.A Additional details of homecare monitoring use case and running example

This addendum includes additional details about the homecare monitoring use case, which is described in Section 3.3, and its running example that is used in the discussion of the DIVIDE methodology in Section 3.4, 3.5 and 3.6.

3.A.1 Semantic representation of use case and running example

This part of the addendum provides additional details of how the homecare monitoring use case and its running example are semantically represented with the Activity Recognition ontology.

- Listing 3.8 gives an overview of all prefixes used in the listings with semantic content in this chapter.
- Listing 3.9 lists some ontology definitions that specify when a showering activity prediction corresponds to the routine of a patient and when it does not, based on the activities defined in this patient's routine. Based on these definitions, a semantic reasoner can define a recognized activity as an instance of either `RoutineActivityPrediction` or `NonRoutineActivityPrediction`. The desired output of the semantic AR service consists of instances of these classes and their relations.
- Listing 3.10 gives an example context description of a patient in the described use case scenario. The current location of this patient in the service flat is the bathroom.
- The description of the HomeLab service flat is given in the instantiated example modules `_Homelab` and `_HomelabWearable` of the DAHCC ontologies. The most relevant descriptions of these modules with respect to the running example are presented in Listing 3.11. As can be observed, for each sensor in the home, the observed properties are defined through the `measuresProperty` object property, and the analyzed entity is specified with the `analyseStateOf` property.

3.A.2 End user definition of running example's DIVIDE query as an ordered collection of SPARQL queries

The DIVIDE query corresponding to the running example is detailed in Section 3.5.1. This query can be defined by an end user as an ordered collection of existing SPARQL

Listing 3.8: Overview of all prefixes used in the listings with semantic content in this chapter

```
# Activity Recognition ontology including DAHCC ontology modules
@prefix KBActivityRecognition:
    <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/KBActivityRecognition/> .
@prefix ActivityRecognition: <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/> .
@prefix MonitoredPerson: <https://dahcc.idlab.ugent.be/Ontology/MonitoredPerson/> .
@prefix SensorsAndActuators: <https://dahcc.idlab.ugent.be/Ontology/SensorsAndActuators/> .
@prefix SensorsAndWearables: <https://dahcc.idlab.ugent.be/Ontology/SensorsAndWearables/> .
@prefix Sensors: <https://dahcc.idlab.ugent.be/Ontology/Sensors/> .

# instances in use case scenario
@prefix : <http://divide.ilabt.imec.be/idlab.homelab/> .
@prefix patients: <http://divide.ilabt.imec.be/idlab.homelab/patients/> .
@prefix Homelab: <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/> .
@prefix HomelabWearable: <https://dahcc.idlab.ugent.be/Homelab/SensorsAndWearables/> .

# SAREF and extensions
@prefix saref-core: <https://saref.etsi.org/core/> .
@prefix saref4ehaw: <https://saref.etsi.org/saref4ehaw/> .
@prefix saref4bldg: <https://saref.etsi.org/saref4bldg/> .
@prefix saref4wear: <https://saref.etsi.org/saref4wear/> .

# other imports
@prefix time: <http://www.w3.org/2006/time#> .
@prefix eep: <https://w3id.org/eep#> .

# generic prefixes
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .

# definitions within DIVIDE
@prefix sd: <http://idlab.ugent.be/sensdesc#> .
@prefix sd-query: <http://idlab.ugent.be/sensdesc/query#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
```

Listing 3.9: Example of how different subclass and equivalence relations between concepts are defined in the `KBActivityRecognition` ontology module of the Activtiy Recognition ontology, allowing a semantic reasoner to derive whether an activity prediction corresponds to a person's routine or not. To improve readability, all definitions are listed in Manchester syntax and the `KBActivityRecognition:` prefix is replaced by `:`.

```
:RoutineActivityPrediction SubClassOf: ActivityRecognition:ActivityPrediction
:NonRoutineActivityPrediction SubClassOf: ActivityRecognition:ActivityPrediction

:RoutineShoweringActivityPrediction SubClassOf: :RoutineActivityPrediction
:RoutineShoweringActivityPrediction EquivalentTo:
    :RoutineActivityPrediction and :ShoweringActivityPrediction
:RoutineShoweringActivityPrediction SubClassOf: :ShoweringActivityPrediction
:RoutineShoweringActivityPrediction EquivalentTo:
    :ShoweringActivityPrediction and
    (ActivityRecognition:activityPredictionMadeFor some :UserWithShoweringRoutine)

:NonRoutineShoweringActivityPrediction SubClassOf: :NonRoutineActivityPrediction
:NonRoutineShoweringActivityPrediction EquivalentTo:
    :NonRoutineActivityPrediction and :ShoweringActivityPrediction
:NonRoutineShoweringActivityPrediction SubClassOf: :ShoweringActivityPrediction
:NonRoutineShoweringActivityPrediction EquivalentTo:
    :ShoweringActivityPrediction and
    (ActivityRecognition:activityPredictionMadeFor some :UserWithoutShoweringRoutine)

:ShoweringActivityPrediction SubClassOf: ActivityRecognition:ActivityPrediction
:ShoweringActivityPrediction EquivalentTo:
    ActivityRecognition:ActivityPrediction and
    (ActivityRecognition:forActivity some ActivityRecognition:Showering)

:UserWithShoweringRoutine EquivalentTo:
    saref4ehaw:User and
    (MonitoredPerson:hasRoutine some (
        ActivityRecognition:Routine and
        (ActivityRecognition:consistsOf some ActivityRecognition:Showering)))
:UserWithoutShoweringRoutine EquivalentTo:
    saref4ehaw:User and
    (:doesNotHaveActivityInRoutine some ActivityRecognition:Showering)
```

Listing 3.10: Context description of the example patient in the use case scenario and corresponding running example, in RDF/Turtle syntax. Only a selected set of context definitions are presented, some are omitted.

```
# patient with ID 157 lives in a smart home called the HomeLab
patients:patient157 rdf:type saref4ehaw:Patient ;
    MonitoredPerson:livesIn Homelab:homelab .

# patient has a location tag
patients:patient157 rdf:type saref4wear:Wearer .
Homelab:AQURA_10_10_145_9 saref4wear:isLocatedOn patients:patient157 ;
    MonitoredPerson:hasLocation Homelab:homelab .

# patient has a morning routine consisting of a series of activities
patients:patient157 MonitoredPerson:hasRoutine :MorningRoutine_patient157 .
:MorningRoutine_patient157 rdf:type ActivityRecognition:MorningRoutine ;
    ActivityRecognition:consistsOf _:A1, _:A2, _:A3, _:A4, _:A5, _:A6 ;
    ActivityRecognition:nextActivity _:A1 .
_:A1 rdf:type ActivityRecognition:WakingUp ;
_:A2 rdf:type ActivityRecognition:Toileting .
_:A3 rdf:type ActivityRecognition:Showering .
_:A4 rdf:type ActivityRecognition:BrushingTeeth .
_:A5 rdf:type ActivityRecognition:EatingMeal .
_:A6 rdf:type ActivityRecognition:WatchingTVActively .
_:A1 ActivityRecognition:nextActivity _:A2 .
_:A2 ActivityRecognition:nextActivity _:A3 .
_:A3 ActivityRecognition:nextActivity _:A4 .
_:A4 ActivityRecognition:nextActivity _:A5 .
_:A5 ActivityRecognition:nextActivity _:A6 .

# patient is currently located in the bathroom
patients:patient157 MonitoredPerson:hasIndoorLocation Homelab:bathroom .
```

queries. This definition can then be translated by the DIVIDE query parser to its internal representation. This addendum section details this end user definition.

The stream and final queries of the definition are shown in Listing 3.12 and 3.13, respectively. There are no intermediate queries. The context enrichment also consists of an empty set of queries, since the stream query is the first query in the ordered set of SPARQL queries used in the stream reasoning system. However, Section 3.A.3 of this addendum discusses a related DIVIDE query that does include a context enrichment and intermediate queries.

Moreover, the DIVIDE query definition contains one stream window with the following properties:

- Stream IRI: `http://protego.ilabt.imec.be/idlab.homelab`
- Window definition: `RANGE PT?rangeS STEP PT?slideS`
- Default window parameter values: `?range` has a default value of 30, `?slide` has default value 10

This window definition contains the two variable window parameters `?range` and `slide`. The definition of a default value for both window parameters implies that they are not used as static window parameters. This can be confirmed by observing their absence in the WHERE clause of the stream query in Listing 3.12.

Listing 3.11: Context description of the service flat of the example patient in the use case scenario and corresponding running example, in RDF/Turtle syntax. Only a selected set of context definitions are presented, some are omitted.

```
# the HomeLab building consists of a bathroom on the first floor
Homelab:homelab rdf:type saref4bldg:Building .
Homelab:firstfloor rdf:type SensorsAndActuators:Floor ;
    saref4bldg:isSpaceOf Homelab:homelab .
Homelab:bathroom rdf:type SensorsAndActuators:BathRoom ;
    saref4bldg:isSpaceOf Homelab:firstfloor .

# the bathroom contains a Netatmo sensor that measures, among others, relative humidity
<https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78>
    rdf:type Homelab:Netatmo ;
    core:measuresProperty Homelab:org.dyamand.types.airquality.CO2 ,
                          Homelab:org.dyamand.types.common.AtmosphericPressure ,
                          Homelab:org.dyamand.types.common.Loudness ,
                          Homelab:org.dyamand.types.common.RelativeHumidity ,
                          Homelab:org.dyamand.types.common.Temperature ;
    Sensors:analyseStateOf Homelab:bathroom ;
    saref4bldg:isContainedIn Homelab:bathroom .
Homelab:Netatmo rdf:type owl:Class ;
    rdfs:subClassOf saref-core:Sensor .
Homelab:org.dyamand.types.common.RelativeHumidity
    rdf:type SensorsAndActuators:RelativeHumidity .

# the HomeLab consists of a location system that can detect the room in which
# the patient is located based on a tag system
Homelab:AQURA_10_10_145_9 core:consistsOf Homelab:AQURA_10_10_145_9.Tag .
Homelab:AQURA_10_10_145_9.Tag rdf:type saref4bldg:Sensor ;
    Sensors:analyseStateOf Homelab:AQURA_10_10_145_9 ;
    core:measuresProperty Homelab:org.dyamand.aqura.AquraLocationState_Protego_User .
Homelab:org.dyamand.aqura.AquraLocationState_Protego_User
    rdf:type SensorsAndActuators:Localisation .
```

In addition, the DIVIDE query definition contains the following solution modifier: `ORDER BY DESC(?t) LIMIT 1`. This solution modifier contains the unbound variable name `?t`, which is allowed since it is present in a stream graph of the stream query (Listing 3.12, line 14).

The variable mapping of stream to final query consists of the stream query variables `?activityType`, `?patient`, and `?model`, which are all mapped to the same variable name in the final query. In addition, it contains the mapping of the variable `?now` in the stream query to the variable `?t` in the final query. The reason for this final mapping becomes clear when inspecting the corresponding generic RSP-QL query pattern in the internal representation of this DIVIDE query (Listing 3.3, lines 43–57): the literal object of the `hasTimestamp` property in the resulting query output indeed corresponds to the `?now` variable.

3.A.3 Additional use case examples associated with running example

The running example of the homecare monitoring use case focuses on the detection of in-home activities that are part of the patient's routine. However, this example does not cover three aspects of the DIVIDE query derivation: the associated DIVIDE query does not have context-enriching queries, no intermediate queries, and no definitions of variable window parameters. Therefore, this addendum zooms in on those aspects for two DIVIDE queries that relate to the DIVIDE query of the running example.

3.A.3.1 Additional example 1: query detecting activities not in the patient's routine

The first additional example focuses on the DIVIDE query that performs the monitoring of the showering activity rule in case the activity is *not* part of the patient's routine. This DIVIDE query is very similar to the DIVIDE query of the running example. However, the output of this DIVIDE query should contain instances of the class `NonRoutineActivityPrediction`. From the ontology definitions in Listing 3.9, it follows that the derivation of such instances requires the association between patient and activity with the `doesNotHaveActivityInRoutine` property for every activity type that is not in the patient's routine. However, such definitions are not present in the regular patient context described in Listing 3.10. Hence, in an existing stream reasoning system applying the DIVIDE query's equivalent as a set of ordered SPARQL queries, the evaluation of the stream query would be preceded by an additional query that is enriching the context with this information. In the DIVIDE query definition, this first SPARQL query would be defined as a context-enriching query. It is presented in Listing 3.14 for illustration purposes.

Listing 3.12: Stream query of the end user definition of the DIVIDE query of the running example that performs the monitoring of the showering activity rule

```
CONSTRUCT {
    _:p rdf:type ActivityRecognition:ActivityPrediction ;
        ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
        ActivityRecognition:activityPredictionMadeFor ?patient ;
        ActivityRecognition:predictedBy ?model ; saref-core:hasTimestamp ?now .
}
FROM NAMED <http://protego.ilabt.imec.be/idlab.homelab>
FROM NAMED <http://protego.ilabt.imec.be/context>
WHERE {
    BIND (NOW() as ?now)

    GRAPH <http://protego.ilabt.imec.be/idlab.homelab> {
        ?sensor saref-core:makesMeasurement [
            saref-core:hasValue ?v ; saref-core:hasTimestamp ?t ;
            saref-core:relatesToProperty ?prop_o ] .
    }

    GRAPH <http://protego.ilabt.imec.be/context> {
        ?model rdf:type ActivityRecognition:ActivityRecognitionModel ;
               <https://w3id.org/eep#implements> [
                   rdf:type ActivityRecognition:Configuration ;
                   KBActivityRecognition:containsRule ?a ] .
        ?a rdf:type KBActivityRecognition:ActivityRule ;
           ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
           KBActivityRecognition:hasCondition [
               rdf:type KBActivityRecognition:RegularThreshold ;
               KBActivityRecognition:isMinimumThreshold "true"^^xsd:boolean ;
               saref-core:hasValue ?threshold ;
               Sensors:analyseStateOf [ rdf:type ?analyzed ] ;
               KBActivityRecognition:forProperty [ rdf:type ?prop ]
           ] .

        ?activityType rdfs:subClassOf KBActivityRecognition:DetectableActivity .
    }

    FILTER (xsd:float(?v) > xsd:float(?threshold))

    GRAPH <http://protego.ilabt.imec.be/context> {
        ?sensor rdf:type saref-core:Device ; saref-core:measuresProperty ?prop_o ;
             Sensors:isRelevantTo ?room ; Sensors:analyseStateOf [ rdf:type ?analyzed ] .
        ?prop_o rdf:type ?prop .

        ?prop rdfs:subClassOf KBActivityRecognition:ConditionableProperty .
        ?analyzed rdfs:subClassOf KBActivityRecognition:AnalyzableForCondition .

        ?patient MonitoredPerson:hasIndoorLocation ?room .
    }
}
```

Listing 3.13: Final query of the end user definition of the DIVIDE query of the running example that performs the monitoring of the showering activity rule

```
CONSTRUCT {
    _:p rdf:type KBActivityRecognition:RoutineActivityPrediction ;
        ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
        ActivityRecognition:activityPredictionMadeFor ?patient ;
        ActivityRecognition:predictedBy ?model ; saref-core:hasTimestamp ?t .
}
WHERE {
    ?p rdf:type KBActivityRecognition:RoutineActivityPrediction ;
       ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
       ActivityRecognition:activityPredictionMadeFor ?patient ;
       ActivityRecognition:predictedBy ?model ; saref-core:hasTimestamp ?t .
    ?activityType rdfs:subClassOf KBActivityRecognition:DetectableActivity .
}
```

Listing 3.14: Context-enriching query in the definition of the DIVIDE query that detects an ongoing activity that is not in a patient's routine. It enriches the context with all activity types that are not part of the patient's routines.

```
CONSTRUCT {
    ?p KBActivityRecognition:doesNotHaveActivityInRoutine [ rdf:type ?activityType ] .
}
WHERE {
    ?p rdf:type saref4ehaw:Patient .

    ?activityType rdf:type owl:Class ;
                  rdfs:subClassOf KBActivityRecognition:DetectableActivity .

    FILTER NOT EXISTS {
        ?p MonitoredPerson:hasRoutine ?routine .
        ?routine ActivityRecognition:consistsOf ?routineActivity .
        ?routineActivity rdf:type ?activityType .
    }
}
```

3.A.3.2 Additional example 2: indoor location monitoring query

The second additional example focuses on the DIVIDE query that corresponds to the monitoring of the patient's location in the home. This DIVIDE query includes variable dynamic window parameters and an intermediate query.

Dynamic window parameters The DIVIDE query contains two context-enriching queries that define multiple dynamic window parameters. These queries are shown in Listing 3.15. The dynamic window parameters defined in the output of these queries are constructed based on the current context concerning any ongoing activity for this patient. It makes a distinction between two scenarios: when an activity *not in* the patient's routine is ongoing (first query), and when an activity *in* the patient's routine is ongoing (second query). Note that for the default case when no activity is currently ongoing, no dynamic window parameters are defined: in those cases, default values for the window parameter variables will be substituted as static window parameters. Moreover, note that the two graph patterns in the WHERE clauses of the queries are semantically distinct: there will never be more than one query for which the graph pattern in the WHERE clause has a matching set of variables. This ensures that there is at most one value defined for the two window parameter variables in the enriched context.

Intermediate query The output constructed by the stream query in this DIVIDE query's definition, is the following:

```
?patient MonitoredPerson:hasIndoorLocationString ?v ;
         saref-core:hasTimestamp ?t .
```

The value of `?v` contains the string representation of the indoor location, as measured by the localization system. However, this does not yet define the location with its actual ontology entity IRI. Therefore, an intermediate query could be used to make this translation. This way, the final query can look for the most recent location IRI. The example of such a combination of intermediate and final query is presented in Listing 3.16. Note that it would also be possible and semantically equivalent to integrate the translation done in the intermediate query into the final query. However, for readability purposes, it is often better to have multiple, simpler SPARQL queries like in this example.

Listing 3.15: Context-enriching queries that define dynamic window parameters for the DIVIDE query that performs the monitoring of the patient's location in the home. They define the window parameters of this location query based on the current context about any ongoing activity for this patient that is or is not part of the patient's known routine.

```
# first context-enriching query
CONSTRUCT {
    sd-query:pattern sd:windowParameters (
        [ sd-window:variable "range" ; sd-window:value 30 ; sd-window:type time:seconds ]
        [ sd-window:variable "slide" ; sd-window:value 30 ; sd-window:type time:seconds ] )
} WHERE {
    ?patient rdf:type saref4ehaw:Patient ; MonitoredPerson:livesIn ?home .
    ?prediction1 rdf:type KBActivityRecognition:RoutineActivityPrediction ;
                 ActivityRecognition:activityPredictionMadeFor ?patient .
    FILTER NOT EXISTS {
        ?prediction2 rdf:type KBActivityRecognition:NonRoutineActivityPrediction ;
                     ActivityRecognition:activityPredictionMadeFor ?patient . }
}

# second context-enriching query
CONSTRUCT {
    sd-query:pattern sd:windowParameters (
        [ sd-window:variable "range" ; sd-window:value 5 ; sd-window:type time:seconds ]
        [ sd-window:variable "slide" ; sd-window:value 5 ; sd-window:type time:seconds ] )
} WHERE {
    ?patient rdf:type saref4ehaw:Patient ; MonitoredPerson:livesIn ?home .
    ?prediction1 rdf:type KBActivityRecognition:NonRoutineActivityPrediction ;
                 ActivityRecognition:activityPredictionMadeFor ?patient .
    FILTER NOT EXISTS {
        ?prediction2 rdf:type KBActivityRecognition:RoutineActivityPrediction  ;
                     ActivityRecognition:activityPredictionMadeFor ?patient . }
}
```

Listing 3.16: Example of intermediate query and final query in the end user definition of the DIVIDE query that performs the monitoring of the patient's location in the home. The solution modifier of the final query would be `ORDER BY DESC(?t) LIMIT 1` to retrieve the most recent location only.

```
# intermediate query
CONSTRUCT {
    ?patient MonitoredPerson:hasIndoorLocationOfInterest [
             saref-core:hasValue ?room; saref-core:hasTimestamp ?t ] .
} WHERE {
    ?patient MonitoredPerson:hasIndoorLocationString [
             saref-core:hasValue ?l ; saref-core:hasTimestamp ?t ] .

    ?room rdf:type saref4bldg:BuildingSpace ; rdfs:label ?roomLabel .
    FILTER (xsd:string(?roomLabel) = xsd:string(?l))
}

# final query
CONSTRUCT {
    ?patient MonitoredPerson:hasIndoorLocation ?room .
} WHERE {
    ?patient MonitoredPerson:hasIndoorLocationOfInterest [
             saref-core:hasValue ?room; saref-core:hasTimestamp ?t ] .
}
```

Addendum 3.B Configuration of the DIVIDE implementation

This addendum gives some examples of how our implementation of DIVIDE, which is presented in Section 3.7, should be concretely configured.

- Listing 3.17 shows an example of the JSON configuration of the DIVIDE system.
- Listing 3.18 contains the JSON configuration of the DIVIDE query for the running use case example discussed in Section 3.5.1. In other words, parsing the configured DIVIDE query with the DIVIDE query parser leads to the DIVIDE query goal in Listing 3.2 and the sensor query rule in Listing 3.3.

Listing 3.17: Example JSON configuration of the DIVIDE system

```
{
  "divide": {
    "kb": {
      "type": "Jena",
      "baseIri": "http://protego.ilabt.imec.be/idlab.homelab/"
    },
    "ontology": {
      "dir": "definitions/ontology/",
      "files": [ "KBActivityRecognition.ttl", "ActivityRecognition.ttl", "MonitoredPerson.ttl",
                 "Sensors.ttl", "SensorsAndActuators.ttl", "SensorsAndWearables.ttl",
                 "_Homelab_tbox.ttl", "_HomelabWearable_tbox.ttl",
                 "imports/eep.ttl", "imports/affectedBy.ttl", "imports/cpannotationschema.ttl",
                 "imports/saref.ttl", "imports/saref4bldg.ttl",
                 "imports/saref4ehaw.ttl", "imports/saref4wear.ttl" ]
    },
    "queries": { "sparql": [ "divide-queries/activity-showering.json" ] },
    "reasoner": { "handleTboxDefinitionsInContext": false },
    "engine": {
      "parser": {
        "processUnmappedVariableMatches": false,
        "validateUnboundVariablesInRspQlQueryBody": true
      },
      "stopRspEngineStreamsOnContextChanges": true
    }
  },
  "server": {
    "host": "localhost",
    "port": { "divide": 8342, "kb": 8343 }
  }
}
```

Listing 3.18: End user definition of the DIVIDE query of the running example that performs the monitoring of the showering activity rule. The content of the file named `stream-query.sparql` is presented in Listing 3.12, the content of the file named `final-query.sparql` is presented in Listing 3.13.

```
{
  "streamWindows": [{
    "streamIri": "http://protego.ilabt.imec.be/idlab.homelab",
    "windowDefinition": "RANGE PT?{range}S STEP PT?{slide}S",
    "defaultWindowParameterValues": {
      "?range": "30",
      "?slide": "10"
    }
  }],
  "streamQuery": "stream-query.sparql",
  "finalQuery": "final-query.sparql",
  "solutionModifier": "ORDER BY DESC(?t) LIMIT 1",
  "streamToFinalQueryVariableMapping": {
    "?activityType": "?activityType",
    "?patient": "?patient",
    "?model": "?model",
    "?now": "?t"
  },
  "contextEnrichment": {
    "queries": [], "doReasoning": true,
    "executeOnOntologyTriples": true
  }
}
```

Addendum 3.C Semantic activity rules of the DIVIDE evaluation scenarios

This addendum contains the semantic description of the activity rules used in the evaluation of the DIVIDE system, as presented in Section 3.8.1.4. These rules include a rule for the toileting, showering and brushing teeth activity. They are semantically defined using the Activity Recognition ontology presented in Section 3.3.2, in the `KBActivityRecognition` ontology module. To improve readability, the `KBActivityRecognition:` prefix is replaced by the `:` prefix in all semantic listings of this addendum.

- Toileting: the person present in the HomeLab is going to the toilet if a sensor that analyzes the energy consumption of the water pump has a value higher than 0. This translates into the following activity rule definition:

```
:toileting_rule rdf:type :ActivityRule ;
    ActivityRecognition:forActivity :_Toileting ;
    :hasCondition :toileting_condition01 .

:toileting_condition01 rdf:type :RegularThreshold ;
    :forProperty :_EnergyConsumption ;
    Sensors:analyseStateOf :_Pump ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "1.0E-5"^^xsd:float .
```

- Showering: the person present in the HomeLab bathroom is showering if the relative humidity in the bathroom is at least 57%. This translates into the following activity rule definition:

```
:showering_rule rdf:type :ActivityRule ;
    ActivityRecognition:forActivity :_Showering ;
    :hasCondition :showering_condition01 .

:showering_condition01 rdf:type :RegularThreshold ;
    :forProperty :_RelativeHumidity ;
    Sensors:analyseStateOf :_BathRoom ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "57.0"^^xsd:float .
```

- Brushing teeth: the person present in the HomeLab bathroom is performing the brushing teeth activity if in the same time window (a) the sensor that analyzes the water running in the bathroom sink measures water running, and (b) the activity index value of the person's acceleration (measured by a wearable) is higher than 30. The activity index based on acceleration is defined as the mean variance of the acceleration over the three axes. This translates into the following activity rule definition:

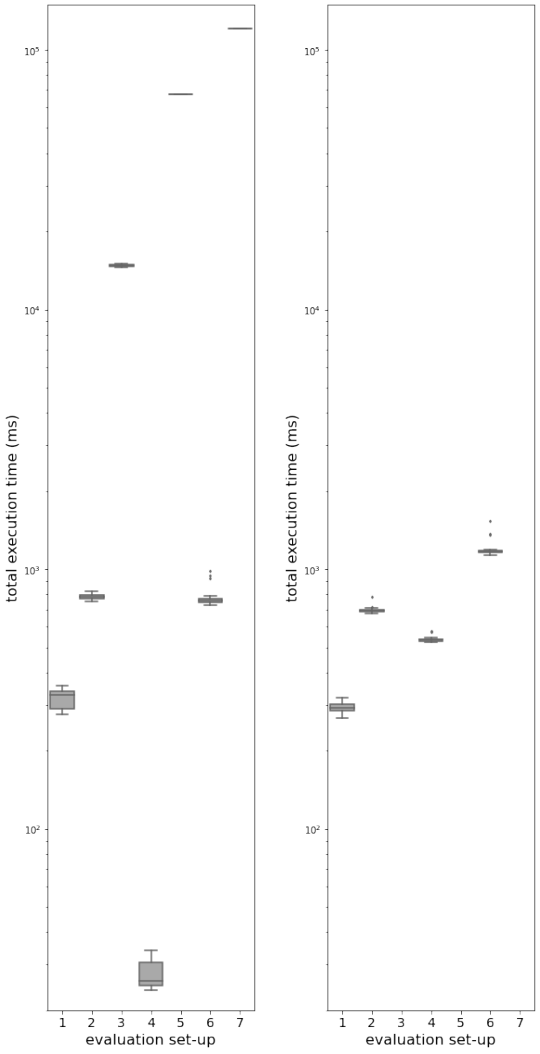
```
:brushing_teeth_rule rdf:type :ActivityRule ;
    ActivityRecognition:forActivity :_BrushingTeeth ;
    :hasCondition :brushing_teeth_condition01 .

:brushing_teeth_condition01 rdf:type :AndCondition ;
    :firstCondition :brushing_teeth_condition02 ;
    :secondCondition :brushing_teeth_condition03 .

:brushing_teeth_condition02 rdf:type :RegularThreshold ;
    :forProperty :_WaterRunning ;
    Sensors:analyseStateOf :_Room ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "1.0E-5"^^xsd:float .

:brushing_teeth_condition03 rdf:type :MeanVarianceThreshold ;
    :forProperty :_WearableAcceleration ;
    Sensors:analyseStateOf :_Patient ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "30.0"^^xsd:float .
```

Addendum 3.D Additional results of the evaluation of DIVIDE in comparison with real-time reasoning approaches

This addendum contains additional results of comparing the real-time evaluation of RSP queries derived by DIVIDE on a C-SPARQL engine, with the other evaluation set-ups that do involve real-time reasoning. These results are complementary to the results shown in Section 3.9.2, for the evaluation set-up as discussed in Section 3.8.3.1.

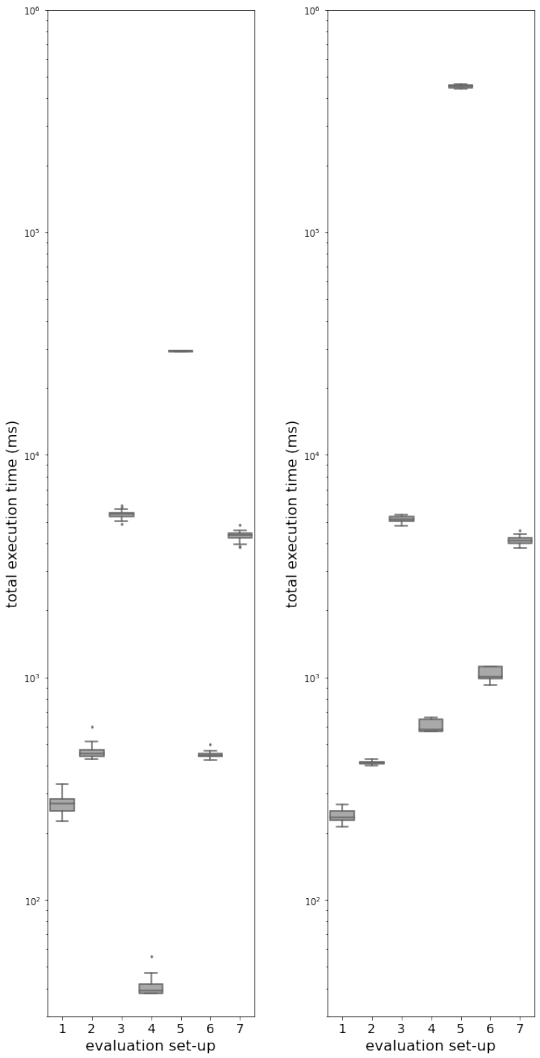
Figure 3.9 includes two boxplots that show the distribution of the total query execution times for the evaluation of the toileting DIVIDE query, for each set-up over the multiple evaluation runs. The distribution is shown for two timestamps corresponding to the mean values that are visualized in the timeline of Figure 3.5. Hence, the distributions correspond to the total execution times measured during the same corresponding evaluation runs. Subfigure 3.9(a) shows the distribution for the total execution times for the event, either streaming or incoming, generated 60 seconds after starting the data simulation. Subfigure 3.9(b) visualizes this distribution for the event generated 1300 seconds after the start of the data simulation. The results show how the non-streaming RDFox set-up has the smallest total execution times in the beginning of the simulation after only 60 seconds, while DIVIDE has smaller total execution times after 1300 seconds. Note that the boxplot distributions after 1300 seconds do not include results for the pipe of C-SPARQL with RDFox set-up (3), the adapted streaming RDFox set-up (5) and the streaming Jena set-up (7) due to those systems running out of memory before reaching this timestamp in the evaluation.

Figure 3.10 shows results completely similar to the results in Figure 3.9, but for the brushing teeth query. The distributions that are visualized correspond to the mean values that are visualized in the timeline of Figure 3.7. Additional results for the showering query are omitted due to their high similarity with the results of the other queries.



(a) At 60 seconds **(b)** At 1300 seconds

Figure 3.9: Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the toileting query. For each evaluation set-up, the results show a boxplot distribution of the total execution time from the generation event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the (final) query. The distribution is shown for two timestamps corresponding to the mean values for this timestamp plotted in Figure 3.5.



(a) At 60 seconds **(b)** At 1300 seconds

Figure 3.10: Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the brushing teeth query. For each evaluation set-up, the results show a boxplot distribution of the total execution time from the generation event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the (final) query. The distribution is shown for two timestamps corresponding to the mean values for this timestamp plotted in Figure 3.7.

4

Towards Knowledge-Driven Symptom Monitoring & Trigger Detection of Primary Headache Disorders

In Chapter 2, a generic cascading reasoning framework was introduced. Chapter 3 presented the semantic IoT platform component DIVIDE, and discussed its methodological design. In this chapter, the cascading reasoning framework with DIVIDE is employed for a new use case compared to the evaluation use cases of the two previous chapters. This way, this chapter demonstrates the generic design of both the cascading reasoning framework and DIVIDE. The use case of this chapter is use case UC3, which is about the continuous follow-up of patients that are diagnosed with a primary headache disorder such as migraine or cluster headache. This use case is associated to the mBrain study. In the chapter, the cascading set-up of the knowledge-driven services of mBrain is presented, in which DIVIDE is used to adaptively monitor contextually relevant headache symptoms and possible headache triggers for such patients. Appendix B further presents detailed background information about the mBrain study for the interested reader.

Since this chapter applies the generic cascading reasoning framework and DIVIDE to another use case, it further discusses research challenge RCH2 ("Adaptive configuration of stream processing queries based on use case context, enabling privacy by design") by addressing research contribution RCO2. Moreover, it also addresses research challenge RCH1 ("Performant & responsive real-time stream reasoning with local autonomy across a heterogeneous IoT network") and its associated research contribution RCO1.

⋆ ⋆ ⋆

M. De Brouwer, N. Vandenbussche, B. Steenwinckel, M. Stojchevska, J. Van Der Donckt, V. Degraeve, F. De Turck, K. Paemeleire, S. Van Hoecke, and F. Ongenae

Published in the Companion Proceedings of the Web Conference 2022, April 2022.

Abstract

Headache disorders are experienced by many people around the world. In current clinical practice, the follow-up and diagnosis of headache disorder patients only happens intermittently, based on subjective data self-reported by the patient. The mBrain system tries to make this process more continuous, autonomous and objective by additionally collecting contextual and physiological data via a wearable, mobile app and machine learning algorithms. To support the monitoring of headache symptoms during attacks for headache classification and the detection of headache triggers, much knowledge and contextual data is available from heterogeneous sources, which can be consolidated with semantics. This chapter presents a demonstrator of knowledge-driven services that perform these tasks using Semantic Web technologies. These services are deployed in a distributed cascading architecture that includes DIVIDE to derive and manage the RDF stream processing queries that perform the contextually relevant filtering in an intelligent and efficient way.

4.1 Introduction

Headache disorders are experienced by many people around the world [1]. Existing headache disorders are classified in the International Classification of Headache Disorders, third edition (ICHD-3) [2]. For each disorder, it defines diagnostic criteria that are the international standard used by doctors in headache diagnosis. Primary headache disorders are those for which the headache and associated symptoms are not a symptom of an underlying disease or condition [2]. Migraine, cluster headache (CH) and tension-type headache (TTH) are the most common primary headache disorders [2].

In current clinical practice, the follow-up of patients with headache attacks happens during a consultation of a patient with his or her doctor. Follow-up and diagnosis of the patient's headache disorder is therefore only based on intermittent subjective data, self-reported by patients during an oral discussion or through existing mobile headache apps such as Migraine Buddy [3]. This current practice is far from optimal. Therefore, the mBrain system [4] tries to move towards a more continuous,

semi-autonomous and objective follow-up of headache patients, based on both self-reported data and objective physiological and contextual data.

The general goal of the mBrain system is to support both doctor and patient in the diagnosis and follow-up of the patient's headache disorder. To this end, data about the patient is collected through different services [4]. Physiological data is collected with the Empatica E4 wearable [5]. This data is consumed by in-house designed machine learning (ML) algorithms that can detect a user's activities, stress periods and sleeping periods. A mobile app allows users to keep a diary of their headache attacks and contextual events (e.g., medicine intakes, food intakes, mood), inspect the ML predictions in a timeline overview, and answer questions about the anticipation of headache attacks, stress, and other events.

Different services can contribute to achieving the goal of the mBrain system. This includes the classification of headache attacks, as well as the detection of potential headache triggers. For the former, knowledge exists from ICHD-3 about diagnostic criteria for the classification of a headache disorder [2]. To this end, relevant data is collected via the mBrain app's diary. Moreover, outputs of the ML algorithms can also detect symptoms relevant for classification. Similarly for trigger detection, lots of data is available from the app and ML algorithms to detect certain triggers. In addition, knowledge on headache triggers of patients is available from the patient himself as well as rule mining services. To perform the given tasks in a context-aware manner, the available data needs to be intelligently consolidated and analyzed. Given the heterogeneous nature of the different sources of knowledge and collected real-time data, semantics are the ideal approach for this [6].

In this chapter, a demonstrator of the knowledge-driven services of the mBrain system is presented. These services are built with Semantic Web technologies, involving the mBrain ontology, RDF stream processing (RSP) and stream reasoning. It includes DIVIDE [7] to derive and manage efficient context-aware stream processing queries. The focus is on the knowledge-driven monitoring of symptoms and other relevant events for headache follow-up and classification, as well as the detection of headache triggers.

4.2 System architecture

The architecture of the knowledge-driven mBrain system follows a cascading reasoning approach [8], since this allows for an efficient distribution of the different semantic tasks across the network. It consists of a local and a central part. An overview of the knowledge-driven mBrain system architecture is shown in Figure 4.1.

The local components should be running on a gateway in the patient's home. The semantic local component is an RDF Stream Processing Engine, filtering any data on its input streams according to the registered continuous queries. These queries are managed by the central DIVIDE component. Events are sent to the processed

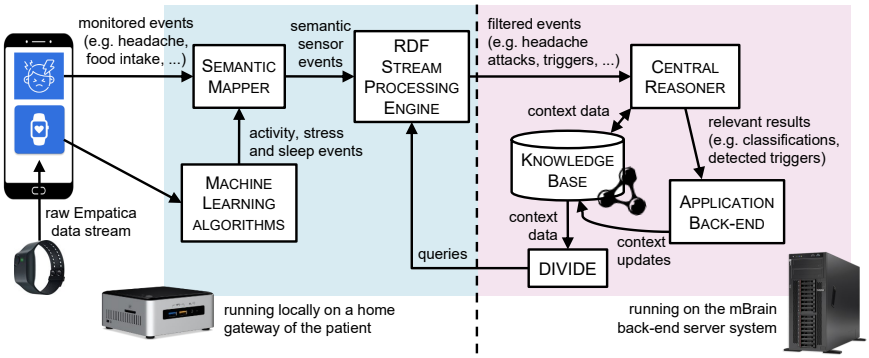


Figure 4.1: Architecture of the knowledge-driven services of the mBrain system

streams by the Semantic Mapper. This component semantically annotates all inputs through the mBrain ontology, which is further discussed in Section 4.3. This mapper receives its inputs from two sources on the patient's smartphone. First, it takes all self-reported events in the mBrain app as input. This includes headache attack registrations, as well as other events such as food intakes. Second, the outputs of the ML algorithms are sent as inputs to Semantic Mapper. These outputs are activity, stress and sleep events predicted based on the raw physiological and accelerometer data collected & streamed by the Empatica E4 wearable over the smartphone.

The central components of the knowledge-driven part of the mBrain architecture are deployed on the mBrain back-end server system. In a real-life scenario, this server system will be hosted by a hospital. This server system contains a Knowledge Base (KB) with the mBrain ontology, including all relevant contextual information of users in the system. This KB is used by the Central Reasoner, a semantic reasoner system that processes the outputs of the local RSP engines. As will be explained in the use case scenarios in Section 4.4.1, this will include ongoing headache attacks that need to be classified by the reasoner, detected triggers and detected symptoms relevant to the classification of a headache attack. Outputs of the Central Reasoner include classification results and detected triggers. They are sent to the Application Back-end, which represents the other non-semantic components in the mBrain system. This component can then act upon the reasoning results, in any implemented way. It could for example generate notifications in the mBrain app or forward the results to a dashboard for the doctor. When contextual information changes from within the Application Back-end, these updates are also forwarded the KB.

DIVIDE [7] is the server component responsible for managing the queries on the local RSP engines. To derive which queries need to be executed, it performs semantic reasoning on the domain knowledge and context relevant to the associated person, which is contained in the KB. It listens to contextual updates in the KB, which trigger the query derivation process. This way, the evaluated RSP queries are always

relevant to the current context, and do not require any more reasoning. DIVIDE fully automates the process of deriving the queries and updating them at the RSP engines.

4.3 mBrain ontology

To achieve the semantic tasks in the mBrain system, the mBrain ontology has been designed [4]. It contains domain knowledge in the headache domain relevant to mBrain. This includes the ICHD-3 classification hierarchy of headache disorders and attacks, and the concepts to semantically describe headache attacks and their properties based on ICHD-3. Moreover, it is connected to the DAHCC (Data Analytics for Health and Connected Care) ontology [9]. This in-house designed ontology has different modules to semantically describe a monitored person, wearables, sensors, and ML predictions.

4.4 Demonstrator

The system architecture described in Section 4.2 is used to perform different knowledge-driven tasks in the mBrain system. This section zooms in on the use case scenarios of some of these tasks that will be the subject of the presented demonstrator. Moreover, an overview is given of any external material relevant to this demonstrator.

In terms of technologies, the RSP engine used within the mBrain system is C-SPARQL [10]. The Central Reasoner and Knowledge Base are deployed with Apache Jena [11].

4.4.1 Use case scenarios

The demonstrator focuses on the three main tasks of the knowledge-driven mBrain system: monitoring of contextual events (symptoms) during headache attacks, monitoring of headache triggers based on user anticipation, and real-time headache classification.

4.4.1.1 Closer monitoring of contextual events during headache attacks

When a patient is experiencing a headache attack, it might be interesting to closely monitor several contextual events such as symptoms associated to the attack. This could for example give relevant insights to validate a headache classification and further refine a patient's diagnosis. To allow this in a semantic system with DIVIDE, the context of a patient in the KB should include information on when a headache attack is occurring, and the patient's (probable) diagnosis. The former can be known through the Empatica E4 wearable which contains a button that patients should push whenever a headache attack is starting.

In the mBrain ontology, performing this monitoring is made possible through the definition of headache attack statistics. The generic query that *could* do such monitoring is presented in Listing 4.1. It monitors any symptom during a headache attack that can be detected by a property associated to an event type in the patient's stream. An example of this is given in Listing 4.2 (lines 2–12): it defines restlessness as a typical associated symptom of cluster headache, which can be detected when an activity event has an activity index value exceeding the defined threshold for restlessness. To only retrieve relevant symptoms, i.e., symptoms associated to the disorder the patient is diagnosed with, semantic reasoning should be done using the definitions in lines 15–24 of Listing 4.2.

In the mBrain system, the generic query in Listing 4.1 is *not* deployed. Instead, with DIVIDE, this generic query can be converted during the query derivation to a simple RSP filtering query yielding a similar blank node of type `RelevantHeadacheAttackStatistic` instead in its CONSTRUCT clause, and with only the triples in lines 36–41 of Listing 4.1 in its WHERE clause. In this query, the query variables `?p`, `?event_type`, `?prop`, `?threshold`, `?symptom`, `?attack` and `?disorder_type` of the WHERE and CONSTRUCT clauses are substituted by DIVIDE during the query derivation. This query would be outputted and registered on the local RSP engine of the patient when the other triples in the WHERE clause of the generic query (lines 14–31) are fulfilled in the patient's context.

4.4.1.2 Monitoring of headache triggers based on user anticipation

Different events can trigger a headache attack. Sources of knowledge on headache triggers for a patient can be the patient himself, or rule mining services that learn the association between headache attacks and contextual events. Some triggers such as stress, physical exercise, sleep deprivation or skipping of meals can be detected by RSP queries combining the domain knowledge and context in the KB with the mBrain event stream. With DIVIDE, specific queries can be defined that detect these triggers. An example for of an RSP query for a patient with a stress trigger is given in Listing 4.3. By using DIVIDE, context-awareness can be easily introduced in this query, e.g., the action state, window size & frequency, or required event duration in this filtering query could be dependent on whether the patient is anticipating an event of the type associated to a known headache trigger for him or her. This anticipation is part of the patient's context in the KB through the mBrain data collection. When a trigger is detected by the local RSP engine, the Central Reasoner could generate a headache alarm and send it to the Application Back-end which can convert it into a mobile mBrain notification.

Listing 4.1: Generic SPARQL query that can detect the occurrence of any headache attack statistic. For simplicity, prefix declarations of the mBrain & DAHCC ontologies are omitted.

```
CONSTRUCT {
  _:a a :HeadacheAttackStatistic ;
      :detectedSymptom ?symptom ;
      saref-core:relatesToProperty [ a ?prop ] ;
      saref-core:hasValue ?v ;
      saref-core:hasTimestamp ?t ;
      :associatedToEventType [ a ?event_type ] ;
      :associatedToHeadacheAttack ?attack ;
      :associatedToDisorder [ a ?disorder_type ] ;
      :associatedToPatient ?p . }
FROM <http://contextaware.ilabt.imec.be/stream>
FROM <http://contextaware.ilabt.imec.be/context.rdf>
WHERE {
  # a patient has a headache attack
  ?p a saref4ehaw:Patient ; :hasHeadacheAttack ?attack .

  # a disorder is defined with an associated symptom
  ?disorder a ?disorder_type ;
            :hasAssociatedSymptom ?symptom .
  ?disorder_type rdfs:subClassOf :HeadacheDisorder .

  # a headache attack symptom can be detected by a
  # threshold on a property associated to an event type
  ?symptom a :HeadacheAttackSymptom ;
      :isDetectedByUpperThreshold [
          a :RegularThreshold ;
          saref-core:hasValue ?threshold ;
          :forProperty [ a ?prop , :EventProperty ;
                         :associatedToEventType
                             [ a ?event_type ] ] ] .
  ?prop rdfs:subClassOf :ConditionableProperty .

  # an event of the given type is present in the
  # patient's event stream, with a value for this
  # property higher than the defined threshold
  ?p saref4ehaw:hasEvent [ a ?event_type ] ;
     :hasAssociatedPropertyValue ?pv ;
     saref-core:hasTimestamp ?t .
  ?pv saref-core:relatesToProperty [ a ?prop ] ;
      saref-core:hasValue ?v .
  FILTER (xsd:float(?v) >= xsd:float(?threshold)) }
```

Listing 4.2: mBrain ontology definitions relevant to the detection of a `RelevantHeadacheAttackStatistic`

```
# Turtle syntax
:_ClusterHeadache a :ClusterHeadache ;
    :hasAssociatedSymptom :Restlessness .
:ClusterHeadache rdfs:subClassOf :HeadacheDisorder .
:Restlessness a :HeadacheAttackSymptom ;
    :isDetectedByUpperThreshold [
        a :RegularThreshold ;
        saref-core:hasValue "5"^^xsd:integer ;
        :forProperty :_ActivityIndex ] .
:_ActivityIndex a :ActivityIndex ;
    :associatedToEventType [ a :Activity ] .
:ActivityIndex rdfs:subClassOf :EventProperty .

# Manchester syntax
:ClusterHeadachePatient ≡ saref4ehaw:Patient and
    :hasHeadacheDisorder some :ClusterHeadache
:ClusterHeadacheAttackStatistic ≡
    :HeadacheAttackStatistic and
    :associatedToDisorder some :ClusterHeadache
:RelevantClusterHeadacheAttackStatistic ≡
    :ClusterHeadacheAttackStatistic and
    :associatedToPatient some :ClusterHeadachePatient
:RelevantClusterHeadacheAttackStatistic ⊑
    :RelevantHeadacheAttackStatistic
```

Listing 4.3: Example RSP query that detects stress as a known trigger for a given patient

```
CONSTRUCT {
    _:a a :HeadacheAlarm ; :relatedDuration ?d ;
        :relatedToTrigger [ a :StressTrigger ] ;
        :targetedAt entity:patient138 . }
FROM STREAM <http://contextaware.ilabt.imec.be/stream>
    [RANGE 60m STEP 5m]
WHERE {
    # patient has a stress event of at least 5 minutes
    entity:patient138
        saref4ehaw:hasEvent [ a DAHCC:Stress ] ;
        saref4ehaw:activityDuration ?d .
    FILTER (xsd:float(?d) >= xsd:float(300)) }
LIMIT 1
```

4.4.1.3 Real-time headache classification

Besides the contextual monitoring use cases described in the previous sections, the knowledge-driven components of mBrain are also responsible for performing real-time classification of headache attacks, based on both information reported by the patient and possible headache attack statistics detected through the RSP queries described in Section 4.4.1.1. Initial versions of semantic queries that classify an individual headache attack as migraine, CH or TTH are constructed based on the diagnostic criteria for these disorders in ICHD-3 [2, 4].

4.4.2 External material

A general video of the mBrain study can be found at https://www.youtube.com/watch?v=wvTY9y-TFZw. It explains the basics of mBrain, allowing for a better understanding of the broader context of its knowledge-driven services. The code of DIVIDE can be found at https://github.com/IBCNServices/DIVIDE. Resource files of the DAHCC ontology, to which the mBrain ontology connects, can be found at https://github.com/predict-idlab/DAHCC-Sources.

4.5 Conclusion

This chapter presents a demonstrator of the knowledge-driven monitoring services used within the mBrain project. mBrain tries to move towards continuous, semi-autonomous, objective follow-up and classification of primary headache disorders based on a combination of self-reported and physiological & contextual data. The architecture of the knowledge-driven mBrain services consists of State-of-the-Art components built on Semantic Web technologies, including DIVIDE to manage the RSP queries that perform the relevant monitoring. This monitoring includes the real-time detection of symptoms during headache attacks, which is useful for classifying and diagnosing headaches, and headache attack triggers.

Funding

This work was partially funded by imec via the AAA Context-Aware Health Monitoring project. N.V. received funding from Ghent University Hospital for his research (Fonds voor Innovatie en Klinisch Onderzoek, 2019). B.S. (1SA0219N) and J.V.D.D. (1S56322N) are funded by a strategic base research grant of Fonds Wetenschappelijk Onderzoek (FWO), Belgium.

Availability of data and materials

Available external data and materials are presented in Section 4.4.2 of this chapter.

References

[1] L. J. Stovner, E. Nichols, T. J. Steiner, F. Abd-Allah, A. Abdelalim, R. M. Al-Raddadi, M. G. Ansha, A. Barac, I. M. Bensenor, L. P. Doan, et al. *Global, regional, and national burden of migraine and tension-type headache, 1990–2016: a systematic analysis for the Global Burden of Disease Study 2016*. The Lancet Neurology, 17(11):954–976, 2018. doi:10.1016/S1474-4422(18)30322-3.

[2] Headache Classification Committee of the International Headache Society (IHS). *The International Classification of Headache Disorders, 3rd edition*. Cephalalgia, 38(1):1–211, 2018. doi:10.1177/0333102413485658.

[3] M. T. Minen, T. Gumpel, S. Ali, F. Sow, and K. Toy. *What are headache Smartphone application (app) users actually looking for in apps: a qualitative analysis of app reviews to determine a patient centered approach to headache Smartphone Apps*. Headache: The Journal of Head and Face Pain, 60(7):1392–1401, 2020. doi:10.1111/head.13859.

[4] M. De Brouwer, N. Vandenbussche, B. Steenwinckel, M. Stojchevska, J. Van Der Donckt, V. Degraeve, J. Vaneessen, F. De Turck, B. Volckaert, P. Boon, K. Paemeleire, S. Van Hoecke, and F. Ongenae. *mBrain: towards the continuous follow-up & headache classification of primary headache disorder patients*. BMC Medical Informatics and Decision Making, 22(1), 2022. doi:10.1186/s12911-022-01813-w.

[5] Empatica. *E4 wristband*, 2020. Accessed: 2020-10-23. Available from: https://www.empatica.com/research/e4.

[6] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. *Semantics for the Internet of Things: Early Progress and Back to the Future*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012. doi:10.4018/jswis.2012010101.

[7] M. De Brouwer, D. Arndt, P. Bonte, F. De Turck, and F. Ongenae. *DIVIDE: Adaptive Context-Aware Query Derivation for IoT Data Streams*. In Joint Proceedings of the International Workshops on Sensors and Actuators on the Web, and Semantic Statistics, co-located with the 18th International Semantic Web Conference (ISWC 2019), volume 2549, pages 1–16, Aachen, 2019. CEUR Workshop Proceedings. Available from: https://ceur-ws.org/Vol-2549/article-01.pdf.

[8] M. De Brouwer, F. Ongenae, P. Bonte, and F. De Turck. *Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions*. Sensors, 18(10):3514, 2018. doi:10.3390/s18103514.

[9] B. Steenwinckel, M. De Brouwer, M. Stojchevska, J. Van Der Donckt, J. Nelis, J. Ruyssinck, J. van der Herten, K. Casier, J. Van Ooteghem, P. Crombez, F. De Turck, S. Van Hoecke, and F. Ongenae. *DAHCC: Data Analytics For Health*

and Connected Care: Connecting Data Analytics to Healthcare Knowledge in an IoT environment, 2022. Accessed: 2022-02-03. Available from: https://dahcc.idlab.ugent.be.

[10] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. *C-SPARQL: a continuous query language for RDF data streams*. International Journal of Semantic Computing, 4(1):3–25, 2010. doi:10.1142/S1793351X10000936.

[11] The Apache Software Foundation. *Apache Jena*, 2021. Accessed: 2022-02-01. Available from: https://jena.apache.org/.

5

Enabling Efficient Semantic Stream Processing across the IoT Network through Adaptive Distribution with DIVIDE

In Chapter 3, the semantic IoT platform component DIVIDE was introduced, which can adaptively update the context-aware queries of stream processing components in an IoT platform based on the changing use case context. This means that up to now, DIVIDE is only adaptive to use case context. However, the situational context in which the queries are deployed is constantly changing. Examples of this include the networking characteristics and the resource usage on local IoT devices. Therefore, this chapter extends the methodological design of DIVIDE by making it adaptive to changing situational context as well. More specifically, it presents how the situational context can be monitored, and how end users can configure how the situational context should influence the configuration of query window parameters and the location in the network to which the queries are distributed. An extended implementation of DIVIDE is discussed, which is evaluated on the homecare monitoring use case UC2, which was introduced in Chapter 3. At the end of this chapter, four addenda are added that contain additional information related to this chapter.

This chapter addresses research challenge RCH3 ("Adaptive configuration and distribution of stream processing queries based on situational context") by discussing research contribution RCO3. It validates research hypothesis RH5: "The methodological design of a semantic IoT platform component that monitors the situational context will result in an adaptive system that can update the window parameter configuration and distribution (i.e., location) to varying

situational context, precisely according to use case specific rules and thresholds as defined by the end user, for a realistic local data stream of at least 150 observations per second.".

⋆ ⋆ ⋆

M. De Brouwer, F. De Turck, and F. Ongenae

Submitted for review to Journal of Network and Systems Management, June 2023.

Abstract
In the Internet of Things (IoT), semantic IoT platforms are often used to solve the challenges associated with the real-time integration of heterogeneous IoT sensor data, domain knowledge and context information. Existing platforms mostly have a static distribution and configuration of queries deployed on the platform's stream processing components. In contrast, the environmental context in which queries are deployed has a very dynamic nature: real-world set-ups involve varying tasks, device resource usage, networking conditions, etc. To solve this mismatch, this chapter presents DIVIDE, an IoT platform component built on Semantic Web technologies. DIVIDE has a generic design containing multiple subcomponents that monitor the environment across a cascading architecture. By monitoring the use case context, DIVIDE adaptively derives the appropriate stream processing queries in a context-aware way. Using a Local Monitor deployed on edge devices, situational context parameters are measured and aggregated. The Meta Model allows modeling these measurements, and meta-information about devices and deployed stream processing queries. Through the definition of application-specific Global Monitor queries that are continuously evaluated centrally on the Meta Model, end users can dynamically configure how the situational context should influence the window parameter configuration and distribution of queries in the network. The chapter evaluates a first implementation of DIVIDE on a homecare monitoring use case. The results show how DIVIDE can successfully adapt to varying device and networking conditions, taking into account the use case requirements. This way, DIVIDE allows better balancing use case specific trade-offs and achieves more efficient stream processing.

5.1 Introduction

The Internet of Things (IoT) is characterized by a high variety of internet-connected devices and sensors that continuously generate and process data. A big advantage of the IoT is that processing devices and applications can easily combine and integrate existing domain knowledge and contextual information with the generated real-time sensor data streams, in order to perform complex processing tasks in a context-aware

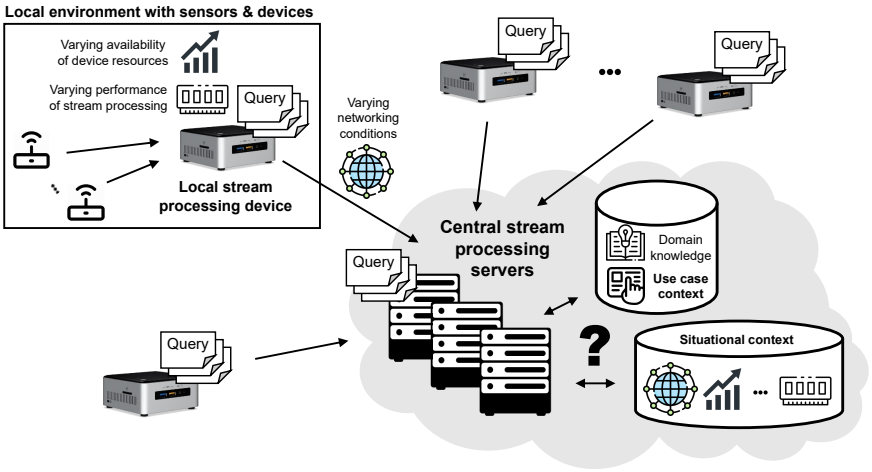


Figure 5.1: Illustration of the static distribution and configuration of stream processing queries across the full network in a semantic IoT platform, in a very dynamic environment. Use case context and various external situational context parameters can vary over time. This requires an adaptive, dynamic distribution and configuration of queries as well, in an efficient way. Achieving this is the main objective of this chapter.

manner [1]. However, this integration of multiple data sources is challenging due to their high volume, variety and velocity [2].

Semantic IoT platforms solve the challenges associated with the real-time integration of IoT sensor data, domain knowledge and context information [3, 4]. They do this by deploying these tasks on a uniform, aggregated data model. Typically, Semantic Web technologies are employed, where this data model is represented by ontologies that formally define the application-specific concepts and their relationships and properties. Stream processing components in the IoT platform continuously evaluate semantic queries on this aggregated data model, using existing stream reasoning techniques [5].

In currently existing semantic IoT platforms, the distribution and configuration of these stream processing and stream reasoning tasks is rather static [3]. This means that the tasks are translated to semantic queries that are deployed across the network's processing components according to a static configuration, i.e., on a fixed location with a fixed set of properties. However, in contrast, the environmental context in which the tasks are deployed has a very dynamic nature. This is true for both use case specific context and the situational context. The latter includes external factors such as network properties, utilization of device resources, properties of the data stream such as the number of events that need to be processed, and the availability of device resources to the stream processing components. Hence, a static distribution and configuration of tasks is not optimal in a dynamic IoT context with a variety of tasks and nodes with varying resources, across a fluctuating network. This is illustrated in Figure 5.1.

To illustrate this with an example, consider a homecare monitoring use case in the healthcare domain, which includes multiple smart homes and an alarm center managing patient calls. In this example, the use case context includes both the Electronic Health Record (EHR) of patients and the location of patients in their smart homes. Both impact which monitoring tasks should be performed, but continuously evolve over time. In addition, a trade-off between cost and patient security needs to be made to optimally distribute the required monitoring tasks across the network. To achieve optimal patient security, all raw sensor data is ideally available on the central servers of the alarm center, in order to motivate decisions and make detailed analyses whenever needed. This requires the processing tasks to be performed centrally. However, to reduce central server costs, less delicate tasks can be executed locally as well, especially when networking conditions do not allow the efficient forwarding of all raw sensor data to the central servers. This is however only feasible if the local devices have enough available resources to perform the processing tasks. Since both network conditions and local device resource usage are situational context parameters that can heavily fluctuate, a static configuration and distribution of the homecare monitoring tasks cannot take all the given requirements into account in a dynamically evolving environment. Instead, a dynamic distribution and configuration of those tasks is required to optimally balance the presented trade-off.

Similarly for other use cases, the dynamic nature of the environmental context is not suited for a static distribution of processing tasks. In the smart cities domain, the network traffic can largely vary over time, highlighting the need for a smart adaptation of task distribution across the network. Moreover, in asset monitoring, adequate follow-up of assets is required without overdoing local device resources, preferring a dynamic configuration of processing task properties such as their execution frequency.

These different examples of various IoT application domains address the key need for the dynamic configuration and distribution of stream processing tasks across the network in a semantic IoT platform. First of all, only relevant tasks should be deployed at all times. Ideally, the location of the deployed individual semantic tasks (queries) can be adaptively shifted between central devices and local & edge devices. Moreover, the semantic queries should also have dynamic properties such as their execution frequency and the size of the data window on which they are executed. All aforementioned decisions should be made dynamically based on the environmental context in which the tasks are deployed. Importantly, as the examples also illustrate, it can differ for different use cases what parameters of the situational context exactly influence this task configuration and distribution, and how.

Hence, in summary, the main research objective of this chapter is to design a semantic IoT platform component that can be deployed in a semantic IoT architecture to dynamically define, distribute and configure the relevant stream processing tasks based on the environmental context. In other words, the designed component will dynamically decide which tasks are executed across the network, where

in the network, and how. It will achieve this by monitoring both the situational and use case context. More specifically, we set the following research objectives for the design of this component:

1. The component should enable the monitoring of various situational context parameters such as the network characteristics, resource usage on the stream processing devices, data stream properties and real-time performance of the stream processing components.

2. The component should allow dynamically updating the location of stream processing queries across the IoT network and the window parameters of the stream processing queries (execution frequency, size of data window), based on the monitored situational context.

3. The component should allow dynamically configuring for each individual use case how the situational context influences the location and/or window parameters of the deployed stream processing queries, in order to optimally balance use case specific trade-offs and achieve efficient stream processing.

4. The component should have a generic design that easily allows monitoring additional properties of the situational context.

5. The component should ensure that only the relevant stream processing queries are deployed on all components of the IoT network at all times, based on existing domain knowledge and the current use case context.

The remainder of this chapter is structured as follows. Important background information is provided in Section 5.2. Section 5.3 presents the full methodology of the research that endeavors to achieve the research objectives of this work. Details on our implementation of this methodology are given in Section 5.4. Moreover, relevant related work for this chapter is presented in Section 5.5. Sections 5.6 and 5.7 describe the set-up and results of the performed evaluations. Section 5.8 further discusses the evaluation results. Finally, Section 5.9 concludes the main findings of the chapter and highlights future work.

5.2 Background

The presented IoT platform component builds further on DIVIDE, which is the result of previous research in our research group [6, 7]. DIVIDE itself is built on existing Semantic Web technologies. This section discusses background information on both aspects that is relevant to the remainder of this chapter.

5.2.1 Semantic Web technologies

The Resource Description Framework (RDF) [8] and the Web Ontology Language (OWL) [9] are two existing standards that allow modeling heterogeneous data sources in a uniform, aggregated data model using ontologies [4]. RDF data is represented as a graph of triples, where every triple connects a subject to an object with a predicate. Different data formats exist to express and store RDF data. Popular examples are RDF/Turtle and N-Triples. Similarly, Manchester syntax is a compact syntax to represent OWL 2 ontologies. The SPARQL Protocol and RDF Query Language (SPARQL) can be used to write and evaluate queries on RDF data [10].

Semantic reasoning is a technique to derive new knowledge from a set of asserted facts and axioms defined in ontologies. Different OWL 2 language profiles exist [11]. Every profile gives an OWL 2 ontology a level of expressivity to define axioms that can be used by a semantic reasoner. The OWL 2 RL profile allows defining axioms that can be evaluated by a rule engine.

Stream reasoning focuses on adopting semantic reasoning techniques for streaming data [5]. RDF Stream Processing (RSP) engines such as C-SPARQL continuously process RDF data streams by evaluating RSP queries [12, 13]. These queries are evaluated on a data window that is put on the data streams. Based on the window parameters of the data windows defined in the query, a window is triggered at specific times to evaluate the query. The window parameters include the size and (sliding) step of the window. The latter defines the period between data window triggers and thus the query execution frequency. RSP-QL is used within DIVIDE for representing RSP queries, as it is a reference model that unifies the semantics of existing RSP approaches [14]. Finally, cascading reasoning is an approach that allows expressive semantic reasoning over high-velocity data streams by introducing a processing hierarchy of reasoners [15–17].

5.2.2 DIVIDE

DIVIDE is a semantic component that can automatically and adaptively derive and manage the relevant queries for the stream processing components in an IoT platform [6]. It does this in a context-aware way, by monitoring the use case context relevant to the different components in the network. Whenever DIVIDE observes a change to the use case context that is relevant to a specific component, it derives the stream processing queries that are contextually relevant given the updated use case context. DIVIDE performs semantic reasoning on the current use case context to derive the relevant queries. This way, DIVIDE ensures that no more real-time reasoning is required while evaluating the resulting stream processing queries. Hence, these queries can be efficiently executed in comparison with real-time reasoning approaches, also on low-end IoT devices with few resources. Moreover, by managing

the stream processing queries in an automated, adaptive and context-aware way, it reduces the manual, labor-intensive effort required to (re)configure those queries.

In its methodological design, DIVIDE uses the rule-based Notation3 (N3) Logic [18], which is a superset of RDF/Turtle [8]. Hence, the semantic reasoner used by DIVIDE supports N3 and can reason within the OWL 2 RL reasoning profile. Moreover, DIVIDE uses the concepts of a DIVIDE component and a DIVIDE query. A DIVIDE component is an entity in the IoT network on which a single RSP engine runs. Every DIVIDE component is linked to several named graphs in the use case context. Updates to this relevant use case context result in a new DIVIDE query derivation for that component, for all DIVIDE queries registered to the system. A DIVIDE query is a generic template definition of an RSP query that should perform a real-time processing task on the RDF data streams generated by the different local components in the system. The internal representation of a DIVIDE query contains a generic RSP-QL query pattern that typically uses generic ontology concepts in its subparts to allow representing multiple possibly contextually relevant tasks at once. Moreover, the internal representation includes multiple semantic rules that are used by the rule reasoner during the query derivation to construct the resulting RSP queries for the DIVIDE component's RSP engine. The first rule is the goal, which defines the semantic output that should be filtered by the resulting RSP queries. The sensor query rule is the main rule of a DIVIDE query and contains a semantic definition of input variables and static window parameters. The input variables are all variables in the generic RSP-QL query pattern that are dependent on the use case context, while the static window parameters can either have default values or be dependent on the use case context as well. During the query derivation, both sets of parameters are substituted in the RSP-QL query pattern for every set of relevant values, based on the current use case context. Hence, DIVIDE also allows updating the window parameters (window size and sliding step) of the deployed stream processing queries.

The DIVIDE query derivation process for a given combination of a DIVIDE component and DIVIDE query consists of different sequentially executed steps. First, the updated context relevant to that component is enriched by executing any defined context-enriching queries on the data model (step 1). These queries are part of the definition of a DIVIDE query, and can extend the context with additional triples. These triples can include dynamic window parameters, which are prioritized in the substitution over static window parameters. Consequently, semantic reasoning is performed on the enriched context and domain knowledge to construct a proof that contains the details of the derived queries and how to instantiate them (step 2). Next, the derived queries are extracted from the proof (step 3) and the instantiated input variables of these derived queries are substituted into the generic RSP-QL query pattern of the DIVIDE query (step 4). Moreover, window parameters are substituted in a similar way (step 5). This window parameter substitution step is split in two parts: first, dynamic window parameters are substituted, *if* present in the enriched context. Second,

static window parameters are substituted for only those window parameter variables that have not yet been substituted. Finally, the resulting RSP-QL queries are translated to the query language of the DIVIDE component's RSP engine and the active, registered RSP queries on this local RSP engine are updated (step 6).

Looking at the research objectives of this work outlined in Section 5.1, it is clear that the first version of DIVIDE resulting from previous research already solves research objective 5: it already derives and manages the RSP queries in an adaptive and context-aware way, based on domain knowledge and use case dependent context. However, DIVIDE cannot yet monitor the situational context and leverage these monitored properties to manage the configuration and distribution of the stream processing queries across the IoT network in an intelligent, dynamic, use case specific way. Moreover, DIVIDE currently always deploys RSP queries associated to a DIVIDE component in the IoT network on the RSP engine that is running on this component's device. It can adaptively update the registered queries, but cannot update the location of the queries by for example moving queries between edge and cloud. Hence, this research focuses on an updated version of DIVIDE, where its design and implementation is updated and extended to fulfill the research objectives of this work.

5.3 Methodology

To achieve the main research objectives of this chapter, the design of DIVIDE is updated to allow performing automated monitoring of the situational context in which semantic queries are deployed across the IoT network, and to allow defining use case specific rules that automatically update the window parameters or distribution of the deployed queries across the network. This section zooms in on the design of DIVIDE by first presenting the overall cascading architecture in which the different subcomponents of DIVIDE are deployed. Moreover, the most important methodological details of these subcomponents are further discussed.

5.3.1 Monitoring architecture

Figure 5.2 provides an overview of the overall architecture of a typical cascading reasoning set-up in an IoT network, in which DIVIDE should be deployed. This architecture is split up in two parts: a central part with components that run centrally in the cloud, and a local part containing components that are deployed on local or edge devices of the IoT network. The central part contains the Central Processing Component, the Knowledge Base and DIVIDE Central. The local part contains the DIVIDE Local Monitor, as well as a Semantic Mapper and a Local (or Edge) RSP Engine. In a typical IoT network, multiple devices exist that contain the local components. In the context of DIVIDE, there is one set of local components for every DIVIDE component. In contrast, there is only one set of central components.

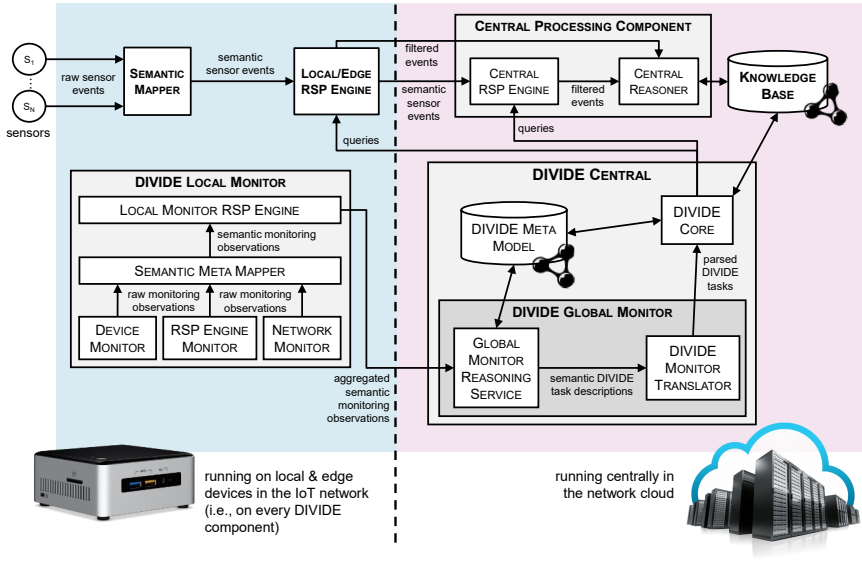


Figure 5.2: Overview of the different subcomponents of DIVIDE in the architecture of a typical cascading reasoning set-up in an IoT network

Centrally, the Knowledge Base contains the semantic representation of all domain knowledge and use case context in the system. This data is semantically stored in an RDF-based knowledge graph. Moreover, the components can be clearly split in two groups: the components in the semantic data processing flow (Semantic Mapper, Local/Edge RSP Engine, Central Processing Component) and the subcomponents of DIVIDE (DIVIDE Central, DIVIDE Local Monitor). Both groups are discussed in the following subsections.

5.3.1.1 Semantic data processing flow

The upper part of Figure 5.2 demonstrates how the data flows through the different components, following a cascading reasoning approach. On every DIVIDE component, different sensors generate raw sensor events. These observations are semantically annotated and forwarded as semantic RDF events to the data streams that are registered to the Local RSP Engine. Depending on the query distribution for the corresponding DIVIDE component, the Local RSP Engine can perform different tasks. This distribution is managed by DIVIDE. For every DIVIDE query, the Local RSP Engine can either continuously evaluate the RSP queries derived from that DIVIDE query, or forward the semantic sensor events on the streams to the Central Processing Component. The latter applies whenever DIVIDE decides that the RSP queries derived from a DIVIDE query should be deployed centrally. In that case, the Central

RSP Engine receives all semantic sensor events forwarded by the Local RSP Engine on its data streams, and continuously evaluates the derived RSP queries.

The Local RSP Engines of the different DIVIDE components and the Central RSP Engine all forward the filtered events in the RSP query outputs to the Central Reasoner. This Central Reasoner is the final component in the semantic data processing flow. It is responsible for further processing these events and acting upon them, depending on the use case requirements. To do so, it can interact with all domain knowledge and contextual data in the Knowledge Base. Moreover, the Central Reasoner can also update any relevant use case context in the Knowledge Base.

5.3.1.2 DIVIDE subcomponents

The updated design of DIVIDE contains multiple subcomponents. On every DIVIDE component in the platform, a DIVIDE Local Monitor is deployed. Moreover, on the central server, the DIVIDE Central component is active, which consists of three main entities: DIVIDE Core, the DIVIDE Meta Model and the DIVIDE Global Monitor.

The DIVIDE Core component represents the first version of DIVIDE resulting from our previous research, as discussed in Section 5.2.2. Hence, it is responsible for monitoring contextual changes in the Knowledge Base and triggering the query derivation whenever changes relevant to a DIVIDE component are observed. In addition, the design of DIVIDE Core is updated to allow it to modify the distribution of queries and configuration of query window parameters through DIVIDE tasks forwarded by the DIVIDE Global Monitor. Hence, DIVIDE Core is responsible for managing the queries of both the Central RSP Engine and all Local RSP Engines in the IoT platform.

The DIVIDE Meta Model is an additional internal RDF-based knowledge graph maintained by DIVIDE Core. It contains the Meta Model ontology that allows modeling all relevant meta-information about DIVIDE. This meta-information includes the different DIVIDE components and DIVIDE queries in the system, and the current configuration and distribution of the RSP queries across the network. Moreover, the Meta Model ontology enables the monitoring subcomponents of DIVIDE to model all monitoring observations. The aggregation of all this data in the Meta Model is essential in the design of DIVIDE, since it allows using the monitoring information to manage the configuration and distribution of RSP queries, taking the current configuration and distribution together with the other meta-information into account.

On every DIVIDE Component, a DIVIDE Local Monitor is deployed to perform the actual monitoring of the situational context. This DIVIDE Local Monitor contains multiple individual monitors that each continuously monitor specific parameters on the given component. In its current design, DIVIDE supports the monitoring of network properties through the Network Monitor, device resource usage and availability through the Device Monitor, and characteristics and performance of the Local RSP Engine through the RSP Engine Monitor. All monitoring observations

Listing 5.1: Overview of all prefixes used in the listings with semantic content in this chapter, and in the DIVIDE Meta Model ontology overview in Figure 5.3

```
# DIVIDE Meta Model ontology modules
@prefix divide-core: <https://divide.idlab.ugent.be/meta-model/divide-core/> .
@prefix monitoring: <https://divide.idlab.ugent.be/meta-model/monitoring/> .

# existing, imported ontologies
@prefix saref-core: <https://saref.etsi.org/core/> .
@prefix om: <http://www.ontology-of-units-of-measure.org/resource/om-2/> .

# generic prefixes
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

are semantically annotated using the Meta Model ontology, aggregated by the Local Monitor RSP Engine, and forwarded to the DIVIDE Global Monitor.

Finally, the DIVIDE Global Monitor consists of a Global Monitor Reasoning Service that processes all aggregated monitoring observations received from the different DIVIDE Local Monitors. By evaluating use case specific Global Monitor queries on the DIVIDE Meta Model enriched with this monitoring data, it decides how the distribution of RSP queries and the configuration of the queries' window parameters is adaptively altered. These decisions are outputted in the form of semantic tasks descriptions, which are parsed by the DIVIDE Monitor Translator to actual tasks executable by DIVIDE Core.

5.3.2 DIVIDE Meta Model

The DIVIDE Meta Model is an RDF-based knowledge graph that contains the Meta Model ontology. This ontology is an OWL ontology consisting of two main modules: `DivideCore` and `Monitoring`. `DivideCore` contains all constructs that allow modeling meta-information about DIVIDE, while `Monitoring` allows representing the properties monitored by the DIVIDE Local Monitor and their observations. The `Monitoring` module builds further on the `DivideCore` module by importing it. This section zooms in on both ontology modules by discussing the concepts and relationships defined in the modules, highlighting imported existing ontologies, and explaining how meta-information and monitoring data can be represented in the Meta Model.

Figure 5.3 presents an overview of the DIVIDE Meta Model ontology. It shows the most important classes and properties in `DivideCore` and `Monitoring`, as well as how the concepts are linked to existing ontologies. For reference purposes, Listing 5.1 gives an overview of all prefixes used in this figure and in all listings with semantic content in this chapter.

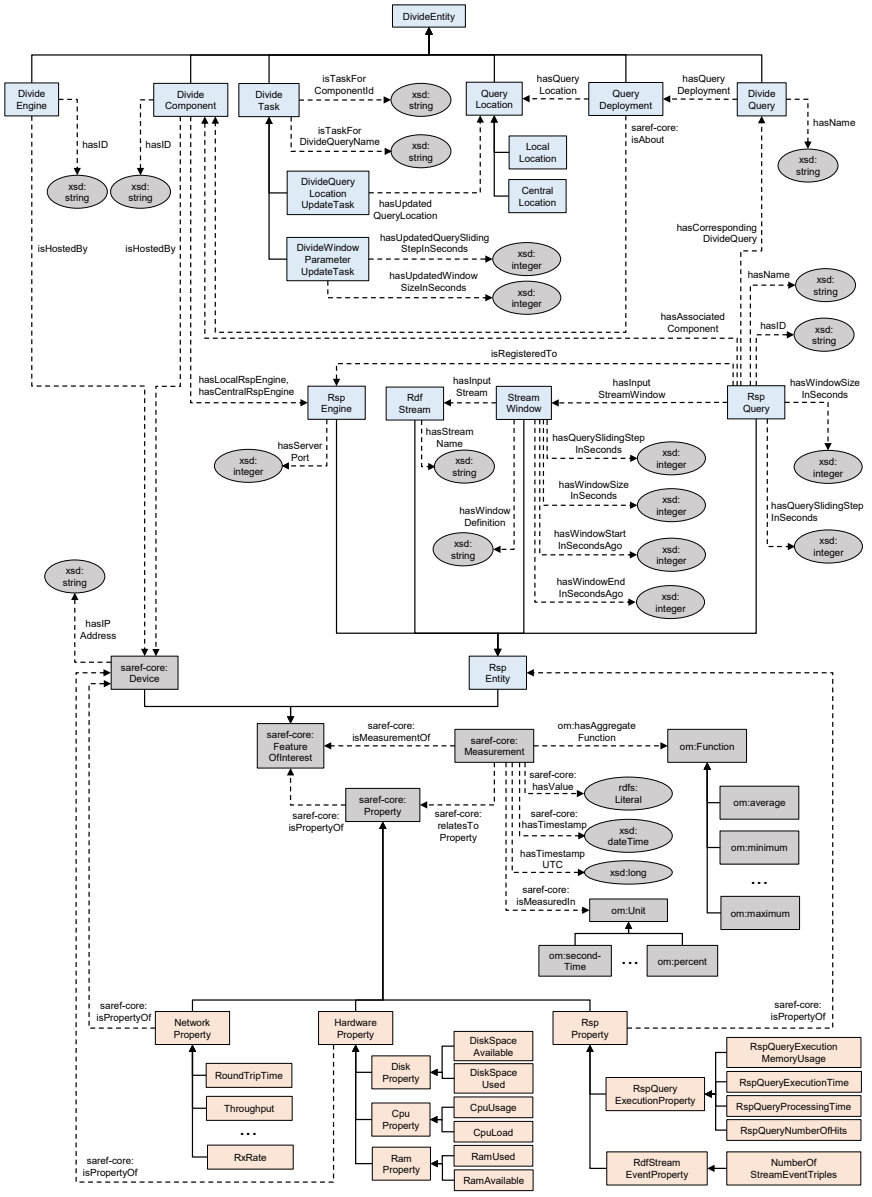


Figure 5.3: Overview of the main concepts in the DIVIDE Meta Model ontology. Items in blue are part of the `DivideCore` module, items in orange are part of the `Monitoring` module, and items in gray are part of imported ontologies. Rectangles represent ontology classes, circles represent literals of the given datatype. Dashed arrows represent object or data properties, the property name is added to each dashed arrow (without prefix for properties in the `DivideCore` and `Monitoring` modules, with prefix for imported ontologies). Full arrows indicate inheritance between classes.

5.3.2.1 Imported ontologies

The DIVIDE Meta Model reuses concepts from existing, well-known ontologies as much as possible. To this end, multiple such existing ontologies are imported by the modules of the Meta Model ontology.

First, the Meta Model ontology imports the existing Smart Applications REFerence (SAREF) ontology [19]. This ontology is an ETSI standard that already defines multiple concepts in the smart applications domain and their relationships and properties. This includes the concept of devices, observable properties, and measurements of those properties in relation to certain features of interest. The latter is especially relevant for the `Monitoring` module.

Second, the Meta Model ontology imports the Ontology of units of Measure [20]. This is an OWL ontology describing the full domain of quantities and units of measure. By integrating this ontology with SAREF in the `Monitoring` module, the quantitative monitoring results can be easily described.

Moreover, several other existing models and ontologies were used as inspiration when designing the overall Meta Model ontology structure. These are discussed in Section 5.5 as related work.

5.3.2.2 `DivideCore` ontology module

`DivideCore` contains all concepts needed to model relevant meta-information about the IoT platform in which DIVIDE is deployed. This includes all properties and relations between DIVIDE entities, as well as the configuration and distribution of RSP queries across the different RSP engines in the network.

To achieve this, `DivideCore` uses the `DivideEntity` and `RspEntity` classes. As shown in Figure 5.3, there are multiple subclasses of `DivideEntity`. In the Meta Model, there is always one `DivideEngine` and typically multiple instances of the `DivideComponent` and `DivideQuery` class.

To keep track of the distribution of `RspQuery` instances across the network, the `QueryDeployment` concept is used. All instances of `RspQuery` that originate from the same `DivideQuery` and are associated to the same `DivideComponent`, have a single `QueryDeployment` instance. This `QueryDeployment` is linked to a `QueryLocation`, which can either be a `LocalLocation` or `CentralLocation`. This represents whether the associated `RspQuery` instances are deployed on the local or central `RspEngine` associated to the `DivideComponent`.

To model the configuration of an `RspQuery`, multiple other subclasses of `RspEntity` exist. An `RspQuery` has one or more `StreamWindow` instances defined as input stream window. Such a `StreamWindow` is linked to an `RdfStream`, and has a certain window definition. This window definition string contains the actual window parameters of the `StreamWindow`. Every `StreamWindow` has a query sliding step, while the description of the other window parameters depends on the exact window

definition string. For example, a window definition `RANGE PT60S STEP PT10S` in RSP-QL translates to value 10 for the `hasQuerySlidingStepInSeconds` property of the `StreamWindow` and value 60 for `hasWindowSizeInSeconds`, while the stream window with window definition `FROM NOW-PT20M TO NOW-PT10M STEP PT2M` leads to value 1200 for `hasWindowStartInSecondsAgo`, value 600 for `hasWindowEndInSecondsAgo`, and value 120 for `hasQuerySlidingStepInSeconds`. Moreover, whenever an `RspQuery` has only one input `StreamWindow`, the values of the `hasWindowSizeInSeconds` and `hasQuerySlidingStepInSeconds` properties of this `StreamWindow` are also linked to `RspQuery` using the same properties. For conformity and uniformity, all window parameters are modeled in seconds.

In the methodological design of DIVIDE, all relevant meta-information of DIVIDE is continuously kept up-to-date in the DIVIDE Meta Model, using the aforementioned concepts in the Meta Model ontology. As this meta-information continuously evolves throughout the runtime of DIVIDE in an IoT platform set-up, one DIVIDE subcomponent is responsible for updating it whenever changes occur. This is DIVIDE Core, as it represents the core subcomponent of DIVIDE that manages the registered DIVIDE queries, DIVIDE components, and derived RSP queries for these DIVIDE components.

Addendum 5.A illustrates with an example how the relevant meta-information of DIVIDE is stored as semantic triples in the DIVIDE Meta Model using the presented concepts of the `DivideCore` ontology module.

Finally, the `DivideCore` ontology module also contains the `DivideTask` class. This class is used by the DIVIDE Global Monitor in the output of the Global Monitor queries to semantically describe the tasks for the DIVIDE engine to update the distribution (query deployment) of the RSP queries associated to a certain DIVIDE component and DIVIDE query, or the configuration of the window parameters of these queries.

5.3.2.3 `Monitoring` ontology module

The `Monitoring` module of the DIVIDE Meta Model ontology is an extension of the `DivideCore` module. It specifically focuses on how the situational context in which DIVIDE operates can be semantically described.

The module heavily uses existing concepts in the imported SAREF ontology to represent individual and aggregated monitoring observations. Every such observation is an instance of the `saref-core:Measurement` class. An instance of `saref-core:Measurement` is related to a certain `saref-core:Property`, and is measured for a specific `saref-core:FeatureOfInterest`. Through the ontology definition, every subclass of `saref-core:Property` can be linked to a specific subclass of `saref-core:FeatureOfInterest` to ensure that measurements of the property are always linked to the given feature of interest type. Moreover, a `saref-core:Measurement` has a value, a timestamp (both as a string and a UTC

millisecond timestamp), and a unit represented with the Ontology of units of Measure. Finally, a measurement can also be defined as an aggregate measurement by linking it to an aggregate `om:Function` such as `om:average`.

The `Monitoring` module defines several subclasses of `saref-core:Property`: `NetworkProperty` for network characteristics, `HardwareProperty` for resource usage and availability of the hardware of devices, and `RspProperty` for specific RSP engine characteristics. Several subclasses of those classes are defined as well. Note that this list could be easily extended with other properties that are also of relevance to be monitored. Moreover, the ontology module also defines that every measurement related to a `NetworkProperty` and `HardwareProperty` is always linked to a `saref-core:Device` as a feature of interest. This will be the device on which the Local Monitor RSP Engine and thus also the Local Monitor is active. For an `RspProperty` measurement, the feature of interest of is always an `RspEntity`. More specifically, stream characteristics (`RdfStreamEventProperty` measurements) and RSP query performance measures (`RspQueryExecutionProperty` measurements) collected by the RSP Engine Monitor are linked to the corresponding RDF streams and RSP queries, respectively.

Addendum 5.A presents an example of how a monitoring observation can be semantically described with the concepts of the `Monitoring` ontology module.

5.3.3 DIVIDE Local Monitor

The DIVIDE Local Monitor performs the actual monitoring of the situational context in which RSP queries are deployed across the IoT network. On every DIVIDE component registered to DIVIDE, a DIVIDE Local Monitor is deployed. A DIVIDE Local Monitor consists of multiple subcomponents: several individual monitors, a Semantic Meta Mapper, and a Local Monitor RSP Engine. The following subsections discuss their design.

5.3.3.1 Individual monitors

The design of the DIVIDE Local Monitor deliberately decouples the individual monitors from the semantic components. This way, the individual monitors can be implemented independently. This modular design allows easily updating or replacing the implementation of the individual monitors, without having to modify any of the other components.

The current methodological design of the DIVIDE Local Monitor includes three individual monitors: a Network Monitor, a Device Monitor and an RSP Engine Monitor. They all monitor important situational context information that is of relevance in multiple IoT application domains when deciding on the distribution of RSP queries across an IoT network and the configuration of their window parameters, as indicated by the examples in Section 5.1.

The purpose of the Network Monitor is to keep track of relevant characteristics about the available network. This is relevant for the query distribution across the network, so that the data flow over the network balances the use case specific requirements with the networking conditions. Therefore, the network monitor should focus on the characteristics of the networking link that connects the DIVIDE component's device (i.e., the device on which the Local Monitor is running) with the central server on which the Central Processing Component is deployed. Potentially relevant networking properties to be monitored include network round-trip time (RTT), throughput, latency, available bandwidth, incoming networking packets received vs. dropped, outgoing networking packets sent vs. dropped, delay, jitter, etc.

The goal of the Device Monitor is to analyze the used and available resources of the local or edge device on which the Local Monitor is running. This can be important information in a use case to decide whether specific RSP queries can be deployed on the Local RSP Engine, or whether they should be moved to the Central RSP Engine. For example, RSP queries operating on large data windows might require a certain quantity of Random Access Memory (RAM) to be available, while other queries with a high execution frequency require a low average Central Processing Unit (CPU) load. Thus, relevant device resource properties to monitor include current CPU usage (either per individual CPU core or overall), CPU average load over a certain time period, used vs. available physical RAM and swap memory, used vs. available disk storage, and possibly others.

Finally, the RSP Engine Monitor is included in the design of the Local Monitor to monitor data stream characteristics and the performance of the continuous query execution on the Local RSP Engine. Stream characteristics include the number of triples per stream event or the number of triples sent on the data stream per time unit. Examples of possibly relevant performance metrics are the execution time of RSP queries, the processing time of these queries (i.e., the time from the query's data window trigger until the generation of the query result), the amount of RAM used by the query execution, and the number of query results. Similarly to the other individual monitors, these different monitored properties can influence whether the query window parameters should be modified, or whether a query should be moved to the Central RSP engine. For example, if the processing time of a query exceeds the period between two query executions (i.e., the query's sliding step), the query execution frequency and/or data window size might need to be lowered.

5.3.3.2 Semantic Meta Mapper and Local Monitor RSP Engine

The design of the DIVIDE monitoring subcomponents, and thus also the DIVIDE Local Monitor, is built upon Semantic Web technologies. Therefore, the DIVIDE Local Monitor contains two semantic components: the Semantic Meta Mapper and the Local Monitor RSP Engine.

In the data flow of monitoring observations, the individual monitors forward the raw monitoring observations to the Semantic Meta Mapper. This component is a general semantic mapper that uses the `Monitoring` module of the DIVIDE Meta Model ontology to semantically annotate these raw monitoring observations. These observations are described as measurements of the monitored property, as explained in Section 5.3.2.3.

The Semantic Meta Mapper forwards all semantic monitoring observations to the Local Monitor RSP Engine. The purpose of this RSP engine is to aggregate the different monitoring observations. Possible aggregations are averaging and taking the maximum or minimum value. These aggregations are performed by continuously evaluating one or more aggregation queries and sending the results of these queries to the central Global Monitor. An example of such an aggregation is shown in Listing 5.9 of Addendum 5.A. It is important to note that the aggregation queries are simple filtering queries: they are executed on a data model that only contains the individual semantic monitoring observations, and require no semantic reasoning during their evaluation.

The rationale behind only sending aggregations to the Global Monitor is twofold. First and foremost, this approach limits and controls the number of events sent over the network. This ensures that the Local Monitor will not further stress the network too much in case of congestion. Second, individual outliers in the observations are leveled out by some aggregations such as averaging. This way, a broader view in time on the monitored observations can be considered by the Global Monitor in its decision making, avoiding constant changes in the query distribution and configuration when this is not desirable.

5.3.4 DIVIDE Global Monitor

The DIVIDE Global Monitor is a subcomponent of DIVIDE that is deployed on the central server, together with the DIVIDE Meta Model and DIVIDE Core. It is the actuator of DIVIDE, as it is responsible for making actual decisions that update the distribution or window parameter configuration of the RSP queries deployed across the IoT network. To do this, it intelligently processes the aggregated semantic monitoring observations received from the Local Monitor instances in the platform.

The DIVIDE Global Monitor consists of two subcomponents: the Global Monitor Reasoning Service, and the DIVIDE Monitor Translator. In short, the Global Monitor Reasoning Service continuously executes Global Monitor queries to decide which tasks to update the RSP query distribution or configuration should be performed. These tasks will be semantically described in the output of the Global Monitor queries. In the current methodological design of DIVIDE, two concrete tasks can be defined: a task to update the location in the network where queries are being executed (distribution update), and a task to update the window parameters of RSP

queries (configuration update). The DIVIDE Monitor Translator parses the semantic task descriptions outputted by the Global Monitor queries and translates them to tasks that can then be executed by DIVIDE Core.

This section zooms in on the details of the DIVIDE Global Monitor. First, the general design of the Global Monitor Reasoning Service and the concept of Global Monitor queries is further detailed. Consequently, the two query distribution and configuration update tasks in the design of DIVIDE are discussed. Finally, a user-friendly grammar to specify the Global Monitor queries through actuation rules is presented.

5.3.4.1 Global Monitor Reasoning Service

The Global Monitor Reasoning Service is a stream-based reasoning service that combines rule reasoning with the continuous evaluation of stream processing queries. It maintains a data stream that is continuously receiving the aggregated monitoring observations from the different Local Monitor instances deployed on every DIVIDE component in the network. The reasoning service works with a tumbling window of a configured interval: every interval, a data model is constructed that contains all aggregated monitoring observations in the window. This data model is temporarily added to the DIVIDE Meta Model, which contains an up-to-date version of all relevant meta-information of DIVIDE. This is maintained by DIVIDE Core, as explained in Section 5.3.2.2. This way, the DIVIDE Meta Model contains an aggregated view on this meta-information, combined with the current monitoring data. The Global Monitor Reasoning Service then performs rule reasoning with an OWL 2 RL reasoner on this aggregated data model, and executes all activated Global Monitor queries in a defined order. Query outputs are forwarded to the DIVIDE Monitor Translator, after which the window of monitoring observations is removed from the DIVIDE Meta Model and OWL 2 RL reasoning is again performed.

The Global Monitor queries that are activated on the Global Monitor Reasoning Service need to be configured by the end user of DIVIDE. This is a deliberate design choice, since the conditions of how the system should update the distribution and window parameter configuration of the system, can be very use case specific, as discussed in Section 5.1. Through the design of DIVIDE an end user can define a Global Monitor query to take into account any information that is present in the DIVIDE Meta Model. This includes information about all devices and components in the network, the current configuration and distribution of RSP queries, and any information about the situational context that is captured by the Local Monitors.

A Global Monitor query should be defined by the end user as a regular SPARQL query. To take into account all information in the DIVIDE Meta Model, the concepts of the Meta Model ontology should be used. Similarly, to define the concrete tasks in the output of a Global Monitor query, the `DivideTask` class of the `DivideCore` ontology module can be used, as explained in Section 5.3.2.3. It is important to note

Listing 5.2: Template to specify a query location update task in the output of a Global Monitor query, for a move to the Central RSP Engine

```
[ a divide-core:DivideQueryLocationUpdateTask ;
  divide-core:isTaskForDivideQueryName ?divideQueryName ;
  divide-core:isTaskForComponentId ?componentId ;
  divide-core:hasUpdatedQueryLocation [ a divide-core:CentralLocation ] ]
```

that by design, such a task is always defined for a combination of a DIVIDE component and DIVIDE query. This means that a query location update task or window parameter update task is always performed for all RSP queries derived from the given DIVIDE query, for the given DIVIDE component. Details of how both types of tasks can be defined and how the design of DIVIDE Core is updated to allow performing them, are presented in the following sections.

5.3.4.2 Query location update task

The first task supported in the current methodological design of DIVIDE is a query location update task, which allows updating the RSP query distribution across the network. More specifically, for a given DIVIDE component, this task changes the deployment location of all RSP queries derived from a specific DIVIDE query. Two possible locations exist: local and central. The local deployment location represents the Local RSP Engine on the local or edge device. This is the device associated to the DIVIDE component, on which the Local Monitor is also deployed. The central location represents the Central RSP Engine of the Central Processing Component on the server in the cloud. In the architecture, this is typically the same device on which the Global Monitor Reasoning Service is active.

Definition in Global Monitor query output Listing 5.2 specifies the template of how to define a query location task in the output of a Global Monitor query, using the `DivideCore` module of the Meta Model ontology. It includes the name and ID of the DIVIDE query and component, respectively, and the new query deployment location.

Design changes to DIVIDE Core to perform task To support the execution of a query location update task, the design of DIVIDE Core has been extended. Performing a query location update does not start a query derivation process, since the actual RSP queries that should be evaluated have not changed. Instead, when moving all RSP queries derived from a DIVIDE query from the Local RSP Engine of a certain DIVIDE component to the Central RSP Engine, DIVIDE Core unregisters these RSP queries from the Local RSP Engine. Moreover, new uniquely identifiable data streams are registered to the Central RSP Engine for all data streams defined in the input stream windows of the original RSP queries. DIVIDE Core then instructs

the Local RSP Engine to forward all data on the original streams to those new corresponding streams on the Central RSP Engine. If multiple queries have the same data streams in their input stream windows, DIVIDE Core ensures that the data of one data stream is only forwarded once, to avoid duplicate network traffic. The input stream windows in the original RSP queries are altered to the new input streams, after which these RSP queries with updated stream windows are registered to the Central RSP Engine. Finally, DIVIDE Core ensures that the observers of the outputs of the original RSP queries on the Local RSP Engine are also registered as observers of the outputs of the corresponding new RSP queries on the Central RSP Engine. During the full process of performing the query location update task, DIVIDE Core instructs the Local RSP Engine to buffer all data on the involved data streams, to avoid data loss.

To perform the opposite task of moving RSP queries derived from a DIVIDE query from the Central RSP Engine to the Local RSP Engine of a DIVIDE component, the aforementioned actions are reverted. This means that the RSP queries are unregistered from the Central RSP Engine, the original RSP queries and their observers are registered again to the Local RSP Engine, and the data forwarding is disabled for those local data streams that do not have any associated data streams in the input stream windows of other queries on the Central RSP Engine.

5.3.4.3 Window parameter update task

The window parameter update task is the second task that can be defined in the output of a Global Monitor query. It allows updating the window parameters of the RSP queries that are derived from a DIVIDE query, for a specific DIVIDE component. A window parameter update task does not alter the actual content of the RSP queries or their deployed location, but only modifies the defined window parameters. In its current design, the DIVIDE Meta Model ontology supports updating either the window size of the input stream window of the derived RSP queries, the sliding step, or both at once.

Definition in Global Monitor query output To define a window parameter update task in the output of a Global Monitor query, the `DivideCore` module of the DIVIDE Meta Model ontology should be used. The template for this definition is presented in Listing 5.3. Besides the name and ID to define the respective DIVIDE query and component, an updated value for the window size and/or sliding step of the input stream window of the derived RSP queries can be defined.

As explained in Section 5.3.2.2, if an RSP query only has a single input stream window, the window size and sliding step of this stream window are also semantically linked to the RSP query itself in the DIVIDE Meta Model. If not, no window parameters are associated to the RSP query instance in the Meta Model. Moreover, the Meta Model ontology also links the updated window parameters in the window

Listing 5.3: Template to specify a window parameter update task in the output of a Global Monitor query

```
[ a divide-core:DivideWindowParameterUpdateTask ;
  divide-core:isTaskForDivideQueryName ?divideQueryName ;
  divide-core:isTaskForComponentId ?componentId ;
  divide-core:hasUpdatedQuerySlidingStepInSeconds ?updatedSlidingStep ;
  divide-core:hasUpdatedWindowSizeInSeconds ?updatedWindowSize ]
```

parameter update task description to the name of the DIVIDE query, and not individually to the different stream windows that are part of the DIVIDE query's input. This is shown in the template in Listing 5.3. Hence, in its current design, the Meta Model ontology only supports window parameter update tasks for DIVIDE queries that only have a single input stream window in their RSP-QL query template. It is however easily possible to extend the design of DIVIDE in the future to also support queries with multiple input stream windows.

Design changes to DIVIDE Core to perform task DIVIDE Core is responsible for performing any window parameter update task that is forwarded by the DIVIDE Monitor Translator. Such a task includes a semantic data model that describes the updated window parameters as dynamic window parameters for the query. This description is identical to how dynamic window parameters can be described in the output of context-enriching queries.

To execute a window parameter update task, DIVIDE Core exploits the existing design of the query derivation process that is discussed in Section 5.2. This process consists of different sequential steps. Updating the window parameters of the derived RSP queries can be regarded as a query derivation in which the updated context only differs in the defined dynamic window parameters. Hence, there is no need to perform steps 1 to 4 (context enrichment, semantic reasoning, query extraction and input variable substitution) again. Instead, only step 5 and 6 need to be performed again. First, the new window parameters are substituted into the saved output of the input variable substitution step. By defining the new window parameter values received from the Global Monitor Reasoning Service as dynamic window parameters, the DIVIDE Monitor Translator ensures that those values are substituted first by DIVIDE Core. As a consequence, if a certain window parameter does not receive a new value in the output of a Global Monitor query, the original value will still be substituted instead of being overruled. As a final step in the original query derivation process, the registration of the corresponding queries is updated on the corresponding DIVIDE component's Local RSP engine or the Central RSP Engine, depending on the currently defined query location.

5.3.4.4 User-friendly grammar to specify actuation rules

The design of DIVIDE allows end users to define specific rules of how the distribution or window parameter configuration of RSP queries in the IoT network should be updated according to the situational context. As explained before, such actuation rules can be defined through Global Monitor queries. However, writing those SPARQL queries requires knowledge about Semantic Web technologies. In addition, an end user should also know how the concepts used within DIVIDE are semantically represented in the DIVIDE Meta Model. Hence, users for whom this is too complicated, as they do not have this knowledge or time to acquire it, might prefer a grammar-like syntax to specify the actuation rules. For this, a BNF (Backus–Naur form) grammar could be designed and used, which would then be used by an automatic parser of the Global Monitor to automatically translate the actuation rule into the appropriate SPARQL query for the Global Monitor Reasoning Service.

As an example, consider a Global Monitor query that reduces the window size of all queries on a DIVIDE component with 10%, if the query is running on the component's Local RSP Engine *and* if the available RAM on the local device drops below 20%. Listing 5.4 presents this actuation rule as an actual Global Monitor SPARQL query. Note that, for every DIVIDE query, it calculates the window size reduction from the smallest window size of all RSP queries derived from this DIVIDE query. A mock-up example of how this actuation rule could be represented with such a BNF grammar is shown in Listing 5.5.

5.4 Implementation

To implement the subcomponents in the methodological design of DIVIDE, we have updated our original implementation of DIVIDE [6]. This original implementation includes different modules that together compose the DIVIDE Core component on Figure 5.2. Our updated version of the DIVIDE implementation includes an update to our implementation of the DIVIDE Core module, and an implementation of the DIVIDE Local Monitor and the DIVIDE Global Monitor. This section specifies some details of this implementation.

5.4.1 DIVIDE Local Monitor

The DIVIDE Local Monitor is implemented as an executable Java JAR. This way, it can be independently started on every DIVIDE component.

5.4.1.1 Configuration of the DIVIDE Local Monitor

The DIVIDE Local Monitor should be configured using a JSON file. It defines the ID of the DIVIDE component on which the monitor is running, which individual

Listing 5.4: Example of a Global Monitor query that reduces the window size of all queries on a DIVIDE component with 10%, if the query is running on the component's Local RSP Engine and if the available RAM on the local device drops below 20%

```
CONSTRUCT {
    [ a divide-core:DivideWindowParameterUpdateTask ;
      divide-core:isTaskForDivideQueryName ?divideQueryName ;
      divide-core:isTaskForComponentId ?componentId ;
      divide-core:hasUpdatedWindowSizeInSeconds ?minUpdatedWindowSize ]
}
WHERE {
    { SELECT ?componentId ?divideQueryName
             (MIN(?updatedWindowSize) AS ?minUpdatedWindowSize)
      WHERE {
          ?device a saref-core:Device ;
                  divide-core:hosts ?component .
          ?component a divide-core:DivideComponent ;
                     divide-core:hasID ?componentId ;
                     divide-core:hasLocalRspEngine ?rspEngine .
          ?rspEngine divide-core:hasRegisteredQuery ?rspQuery .
          ?rspQuery divide-core:hasCorrespondingDivideQuery ?divideQuery ;
                    divide-core:hasAssociatedComponent ?component ;
                    divide-core:hasWindowSizeInSeconds ?windowSize .
          ?divideQuery divide-core:hasName ?divideQueryName .

          ?measurement a saref-core:Measurement ;
                       saref-core:hasValue ?avgRamAvailablePercentage ;
                       om:hasAggregateFunction om:average ;
                       saref-core:isMeasuredIn om:percent ;
                       saref-core:relatesToProperty [ a monitoring:RamAvailable ] ;
                       saref-core:isMeasurementOf ?device .

          FILTER (?avgRamAvailablePercentage < xsd:float(20))

          BIND(xsd:integer(FLOOR(xsd:integer(?windowSize) *
                                 xsd:float(0.9))) AS ?updatedWindowSize)
      }
      GROUP BY ?componentId ?divideQueryName }
}
```

Listing 5.5: Mock-up example of how the Global Monitor query in Listing 5.4 could be represented as an actuation rule with a BNF grammar

```
IF DIVIDE_QUERY(?query) = ?divide_query
   AND COMPONENT(?divide_query) = ?component
   AND AVG(RAM_AVAILABLE_PERCENTAGE(DEVICE(?component))) < 20
THEN UPDATE(?divide_query, ?component,
            {"WINDOW_SIZE": FLOOR(WINDOW_SIZE(?query) * 0.9)})
```

monitors need to be activated, the URL of the Global Monitor Reasoning Service to which aggregated measurements should be sent, and some specific properties relevant to the individual monitors. An example of a JSON configuration of the DIVIDE Local Monitor is provided in Addendum 5.B.

5.4.1.2 Implementation of the Local Monitor RSP engine

The Local Monitor RSP Engine is implemented with the C-SPARQL RSP engine [13]. A C-SPARQL aggregation query is deployed that is executed every 20 seconds on a window of 60 seconds on the monitoring data stream, to which all semantic monitoring observations are sent by the Semantic Meta Mapper. This query calculates the average, minimum & maximum of all measured properties, and sends them to the API of the Global Monitor Reasoning Service. This aggregation query is presented in Addendum 5.B.

5.4.1.3 Implementation of the individual monitors

The current Local Monitor implementation includes a first version of the Network Monitor, Device Monitor and RSP Engine Monitor. All individual monitors continuously forward their observations as JSON messages to a generic monitor observer. This observer forwards every received observation to the Semantic Meta Mapper. To correctly link the monitoring observations to their associated features of interest, the implementation ensures that the individual monitors work with the same IDs of devices and RSP entities as the implementation of the DIVIDE Meta Model and DIVIDE Core. This is achieved by the DIVIDE Global Monitor, which manages the configuration and state of the individual Local Monitor instances.

Network monitor The Network Monitor is implemented as a Python script. This implementation serves as a Proof-of-Concept (PoC) that monitors the networking conditions in two ways.

First, the script manages a Bash `ping` process that sends a ping message (echo request) every second to the central server on which the Central Processing Component is running. This way, it continuously measures the RTT of sending a message from the DIVIDE component's device to the central server.

Second, the script uses the cross-platform `psutil` library [21] to monitor system-wide network I/O statistics over 5 second intervals. These statistics include the network Tx and Rx rate (rate of transmitted and received data), the number of packets sent and received, and the number of dropped packets. They are monitored on the network interface that is used by the DIVIDE component's device to communicate with the central server.

Device Monitor The Device Monitor is implemented as a Python script that uses the cross-platform `psutil` library [21]. The script is called every 5 seconds and measures CPU usage and load, memory usage and disk space usage.

RSP Engine Monitor The implementation of the RSP Engine Monitor consists of two parts: a general part included in the DIVIDE Local Monitor JAR implementation, and an external RSP engine specific part.

The RSP engine specific part is required to extract the relevant monitoring information of the semantic data streams and continuous query executions. To facilitate this, we have implemented an RSP engine wrapper that also includes the implementation of the RSP engine's server API. This wrapper is based on the existing RSP Service Interface for C-SPARQL [22]. The wrapper hosts a WebSocket server on which relevant monitoring information is sent as JSON messages. In our implementation, two types of JSON messages are posted: stream events and query executions. A stream event contains the number of triples posted on a certain stream, while a query execution contains all relevant information of the continuous execution of a registered query: memory usage, query execution time, query processing time and the number of query results. Currently, we have implemented the RSP Engine Monitor for the C-SPARQL RSP engine. To retrieve the monitored information of a query execution, we have modified the source code of the C-SPARQL implementation to send the relevant information via callbacks to our wrapper implementation.

The general part of the implementation included in the DIVIDE Local Monitor JAR consists of a WebSocket client that actively keeps an open connection with the WebSocket server hosted by the RSP engine wrapper. It converts JSON messages received over the WebSocket to individual monitoring JSON observations, which comprise the output of the RSP Engine Monitor.

5.4.2 DIVIDE Global Monitor

The DIVIDE Global Monitor is implemented as an additional module of the original, central DIVIDE implementation. It is thus integrated into the executable Java JAR of DIVIDE Central.

5.4.2.1 Configuration of the DIVIDE Global Monitor

The DIVIDE Global Monitor should be configured in the main JSON file that is used for the configuration of DIVIDE Central. Specifically for the monitor, it defines whether the monitoring subcomponents should be active, a path to the built JAR file of the DIVIDE Local Monitor, and a list of files that contain the Global Monitor queries that should be evaluated.

5.4.2.2 Integration of the DIVIDE Meta Model

On implementation level, the DIVIDE Meta Model is part of the Global Monitor Reasoning Service: this reasoning service maintains a data model with ontology axioms extracted from the Meta Model ontology, and all meta-information about DIVIDE Core represented as contextual data triples.

The integration of the DIVIDE Meta Model into the DIVIDE Core subcomponent is implemented through a level of abstraction: the DIVIDE Meta Model exposes an interface to DIVIDE Core that is called for all relevant meta-information updates. These updates include the addition or removal of a DIVIDE query or DIVIDE component, an update to the RSP queries registered to a DIVIDE component, and the update of the query deployment of a DIVIDE query on a DIVIDE component. Upon every update, a collection of triples is constructed based on templates that are predefined based on the Meta Model ontology concepts. These triples are then added to or removed from the data model maintained by the Global Monitor Reasoning Service.

5.4.2.3 Implementation of the Global Monitor Reasoning Service

The Global Monitor Reasoning Service is implemented using the Apache Jena rule reasoner [23]. It continuously executes all registered Global Monitor queries in the order defined in the JSON configuration, on a tumbling data window of 20 seconds on the stream of aggregated observations received from the deployed Local Monitor instances.

5.4.2.4 Management of the DIVIDE Local Monitor instances

The implementation of the DIVIDE Global Monitor is also responsible for managing the configuration and state of the DIVIDE Local Monitor instances deployed on the DIVIDE components in the IoT network. To achieve this, the SSH and SCP protocols are used. To this end, our implementation assumes that every DIVIDE component in the network is reachable and allows incoming SSH connections using SSH public key authentication with a predefined username. Upon start-up of DIVIDE Central, the DIVIDE Local Monitor JAR and its configuration are copied over SCP to every DIVIDE component, and the JAR is started over an SSH connection. This full process is implemented in Python.

5.4.3 DIVIDE Core

The DIVIDE Core component includes the DIVIDE engine, the DIVIDE reasoning module and the DIVIDE server. For details about the original implementation of these modules, we refer to our previous paper on DIVIDE [6]. To accommodate the DIVIDE Core modules for the extension of DIVIDE with the monitoring subcomponents, several updates have been implemented. First, DIVIDE Core is extended

to alert the necessary updates of meta-information to the DIVIDE Meta Model, as explained before. Second, an implementation is provided for the distribution and configuration update tasks to the RSP queries deployed across the network.

The DIVIDE engine maintains a blocking task queue and a dedicated processing thread to execute these tasks, for every registered DIVIDE component. Existing task types are a task to derive the RSP queries for a DIVIDE query whenever a context change relevant to that component is detected, and a task to remove a DIVIDE query from the component. Two additional DIVIDE engine tasks have been implemented that correspond to the tasks forwarded by the DIVIDE Monitor Translator in the methodological design of DIVIDE: a query location update task, and a window parameter update task.

Query location update task To implement the query location update task, the main configuration of DIVIDE Central was updated to include details about how to communicate with the API of the Central RSP Engine. Furthermore, we have implemented an RSP engine wrapper that exposes an API to manage the RSP engine and retrieve information from it. Concretely, this API allows retrieving details about registered streams, queries and their observers. Moreover, it also allows registering and unregistering a data stream, registering and unregistering an RSP query, and registering or unregistering an observer URL to an RSP query. Finally, it supports enabling and disabling the forwarding of all data on a registered stream over a WebSocket connection to a registered data stream of another RSP engine. Note that this wrapper also includes a part of the implementation of the RSP Engine Monitor, as discussed before.

To use our implementation of DIVIDE, we require that all Local RSP Engine instances and the Central RSP Engine are deployed with this wrapper, or at least offer an API with semantically and syntactically equivalent endpoints. To demonstrate and evaluate our system, we have currently integrated the C-SPARQL RSP engine into our wrapper implementation. Other RSP engines can however be easily integrated in the future.

Window parameter update task The implementation of the window parameter update task makes use of the implementation of the DIVIDE query derivation. To implement this, the DIVIDE reasoning module keeps track in memory of intermediate query results for every combination of DIVIDE query and DIVIDE component. Such an intermediate query result contains the output of the input variable substitution step of the DIVIDE query derivation (step 4), which is explained in Section 5.2.2. Whenever a window parameter update task is then executed for a given DIVIDE query and DIVIDE component, the query derivation is started in step 5 with this saved intermediate query result as input.

5.5 Related work

Multiple ontologies exist in the domains of network monitoring [24]. MonONTO is a domain ontology that bridges the domains of network performance monitoring and application quality of service [25]. It specifically focuses on incorporating domain knowledge through inference rules. In addition, a dedicated ontology for traffic monitoring in IP networks was designed within the European project MOMENT [26]. Similarly, the EU-funded NOVI project resulted in multiple ontologies to describe and monitor network resources [27]. Moreover, Silva et al. have presented an ontology that allows defining a network measurement topology and sampling techniques to enable context-aware network monitoring [28]. Multiple other approaches focus specifically on designing an ontology for telecommunications network management and monitoring [29–31].

In the domain of device monitoring, Funika et al. present an ontology-based approach to perform the monitoring of resource usage in multi-scale platforms [32]. Connecting the domains of device monitoring and IoT, Ryabinin et al. demonstrate an ontology-based approach to manage and monitor resource-constrained Edge Computing devices [33]. The Comprehensive Ontology for IoT (COIoT) tries to build an interoperable knowledge base for IoT environments by reusing core concepts from existing ontologies and adds additional concepts to support the monitoring of context and services [34].

To the best of our knowledge, most of the described ontologies are not available through the cited publications or in well-known ontology repositories[1]. Therefore, we have used the concepts and ontology structures described and presented as figures in the cited publications as inspiration to create the DIVIDE Meta Model ontology presented in Section 5.3.2. In this process, we have focused on the ontology structures that were needed in the methodological design of the monitoring subcomponents of DIVIDE, to achieve the research objectives of this work presented in Section 5.1.

Some additional models and ontologies were used as inspiration or direct imports in the designed Meta Model ontology of DIVIDE. The DEN-ng model is a semantic model for the management of computer networks [35], translated to an ontology file by Jeroen Famaey et al. [36, 37]. Moreover, the SAREF ontology is an ETSI standard that focuses on the smart applications domain [19]. In addition, the Ontology of units of Measure defines all possible quantities and units of measures [20]. Finally, the Computer Hardware Components Ontology is a small ontology that defines useful concepts such as monitoring devices, resources, network topologies, network addresses, etc. [38].

Existing research already focuses on performing monitoring of situational context such as network and device conditions, to dynamically distribute knowledge and

[1]The following online repositories were consulted: https://lov.linkeddata.es/dataset/lov/, https://bioportal.bioontology.org/ontologies, https://www.ebi.ac.uk/ols/ontologies, and http://www.sensormeasurement.appspot.com/?p=ontologies.

processing tasks across the network. Keeney et al. have presented an approach that tries to automatically decentralize the number of required semantic reasoning tasks on semantically enriched data within a network [39]. Similarly, they have presented an approach that efficiently tries to distribute heterogeneous knowledge [40]. Multiple, non-ontology based solutions in this domain exist as well. AIOLOS is a mobile middleware framework that considers the resources of the server and the conditions of the network to determine at runtime whether some tasks of a mobile application need to be offloaded to a nearby server in the network [41]. Moreover, Sebrechts et al. have presented a fog native architecture that intelligently decides how microservice-based applications can be distributed over a network [42]. This approach takes into account device and network conditions, as well as meta-information about the available infrastructure, end user requirements, application tasks and more, in order to improve overall performance according to those application conditions. This way, it combines the advantages of edge computing and cloud native microservice applications. Furthermore, Idrees et al. have designed a protocol that intelligently assigns tasks to edge devices to minimize the network communication costs and the energy usage of edge devices [43].

5.6 Evaluation set-up

To validate and demonstrate our implementation of the methodological design of DIVIDE, it is evaluated on a homecare monitoring use case. This section zooms in on this use case, the compared technical set-ups, and the different scenarios of the evaluation.

5.6.1 Evaluation use case

The evaluation is performed on a homecare monitoring use case in healthcare, which is a well-known IoT application domain [44]. This section zooms in on this use case by describing it, and discussing the ontology, use case context and DIVIDE query that are considered for the evaluation. Furthermore, the section explains which dataset is used for the simulation of realistic homecare IoT data in the evaluation scenarios.

5.6.1.1 Use case description

This section discusses three relevant aspects of the homecare monitoring use case: its storyline, the technical set-up, and the specific homecare monitoring task that is considered in the evaluation scenarios.

Storyline Consider a homecare monitoring IoT environment with an alarm center that is responsible for the monitoring of different service flats spread out across the

city. Every service flat is equipped with a wide range of monitoring sensors and devices. Moreover, every patient is wearing a wearable to monitor the patient's in-house location, acceleration and physiological parameters such as heart rate. Furthermore, a nurse call system is installed in every service flat. This nurse call system allows patients to generate an alarm to the alarm center whenever they are in need in of assistance. A patient can do this by pushing a button on a dedicated wearable device. This alarm is then received by a team of human call operators in the alarm center, who should decide which intervention strategy is required. Possible intervention strategies are calling an ambulance, sending a doctor or a nurse with a certain priority, or calling an informal caregiver to pay a visit to the patient.

To help the human operators with choosing the most optimal intervention strategy, the homecare monitoring installation can be used. In this system, lots of individual measurements are generated by the installed lifestyle monitoring devices, environmental sensors, wearable sensors, and possibly others. By reasoning on these measured parameters in combination with existing medical domain knowledge and use case specific context information such as the patient's disease profile, detailed insights can be generated. Examples of relevant insights that help the human call operators are the activity level of the patient, performed in-home activities, medical conditions, and many others.

Whenever a call is generated by a patient, detailed dashboards should be available to the call operators that can be analyzed to correctly assess the situation. These dashboards are also relevant to the other healthcare professionals, such as the nurses that might be called to visit the patient. Importantly, the dashboards should not only show the insights generated by the different algorithms, but they should also include timeline visualizations of any relevant raw sensor data. To achieve this, as much of the raw sensor data as possible should be available on the central servers of the alarm center. This server-side availability of raw data is especially important for patient security as well, for two main reasons. First, the raw data can be used to motivate why certain decisions were made by the call operators. Second, whenever an intervention is chosen by a call operator that later turns out to be the wrong choice, the raw data also allows analyzing in detail why the wrong intervention was chosen.

Importantly, the use case requirement of patient security should be well-balanced with cost. Sending over all raw sensor data from all service flats to the central server to run all the processing tasks centrally, would require a high-end server infrastructure and thus incur high costs. If the budget does not allow these costs, overusing the existing server-side resources would imply a risk of the server going down and being unavailable at times. For obvious reasons, this is unacceptable in the considered healthcare context. Hence, to reduce costs and ensure the system is not at risk of going down, less delicate tasks for which the central need of raw sensor data is less high, should be executed locally instead.

Technical set-up The set-up of the homecare monitoring system uses the cascading reasoning architecture presented in Section 5.3.1. This architecture involves the deployment of the different local and central subcomponents of DIVIDE. The different homecare monitoring tasks are deployed as RSP queries across the network with DIVIDE by defining them as DIVIDE queries. The central components are running in a server environment in the facilities of the alarm center. The edge components are deployed locally in the service flats of the patients, on the available devices on which the running nurse call system is deployed. In other words, there is one DIVIDE component per patient (i.e., per service flat), with a single instance of the Local RSP Engine and the DIVIDE Local Monitor running on the local nurse call system device.

Concerning the deployed nurse call system, different types and versions of the software exist. Some include more services than others, implying that a different amount of resources is required to run the nurse call system across the different service flats. Hence, different local devices are used to deploy the nurse call system, with a different amount of resources. Since the DIVIDE Local Monitor and Local RSP Engine are also deployed on these devices, the system should be able to adapt to this resource variability in a flexible way.

As the service flats are spread out over the city, public networks are being used for the communication between the service flats and the server infrastructure of the alarm center. This implies that the communication is prone to varying networking conditions over time and across the different service flats. Hence, DIVIDE should take these networking conditions into account when balancing the aforementioned trade-off between patient security and cost. Especially when the network is too slow to allow efficient forwarding of raw sensor data to the central servers, more homecare monitoring tasks might need to be deployed locally. This is obvious: having up-to-date aggregated insights about the patient without being able to inspect the raw data, is still a better situation for the alarm center compared to receiving no up-to-date insights at all.

Evaluation homecare monitoring task This evaluation focuses on one specific part of the in-home monitoring of patients: monitoring the patient's level of activity. This is an important monitoring task for a variety of medical diseases and medical conditions. This includes fall-prone patients, heart patients and patients with dementia. Depending on the condition, the level of granularity that is required for insights into the activity level of the patient differs. The required granularity level influences the priority of the need for the server-side availability of the raw sensor data. For fall-prone patients, fine-grained insights into the activity level at every point in time are required, to precisely detect whenever this patient would fall. For heart patients, some level of granularity is also desirable, as heart conditions might vary over time. For patients with dementia, less granularity is needed, as the healthcare professionals are mostly interested in knowing whether the patient is still moving over time or not.

In this evaluation, the activity level of the patients is measured by calculating the activity index value of the patient's acceleration, which is continuously measured by the patient's wearable. This index is defined as the mean variance of the acceleration over the three axes [45]. The higher this value, the more active the person has been in the considered time window.

5.6.1.2 Ontology, use case context and DIVIDE query

The ontology used for the evaluation is an additional module built upon the Data Analytics for Health and Connected Care (DAHCC) ontology [46]. This DAHCC ontology is a publicly available, in-house designed ontology with different modules that allow connecting data analytics to healthcare knowledge in an IoT environment. It connects several existing ontologies in these domains such as SAREF [19], the SAREF extension for the eHealth Ageing Well domain (SAREF4EHAW) [47], and the Execution-Executor-Procedure (EEP) ontology [48]. Through different models, the DAHCC ontology allows capturing metadata about IoT sensors and observations, different AI algorithms, insights about patient health derived from those algorithms, and how these insights are related to the medical condition of the patients.

The additional module built upon the DAHCC ontology contains a description of a health parameter calculator system. Such a system can be configured with different rules about how to measure certain health parameters. The relevance of these parameters can then be linked to the medical condition of the patients. Specifically for the homecare monitoring task of this evaluation, the ontology defines that the activity index is a relevant health parameter that needs to be monitored for patients that require movement monitoring. This movement monitoring requirement is defined for several medical conditions, such as being fall-prone, being a heart patient, or having dementia.

The use case context for the evaluation scenarios contains a patient diagnosed with one of the aforementioned three medical conditions. This patient is living in a smart home that contains a wide range of IoT sensors, and has a wearable that at least measures 3-axis acceleration. Throughout the course of the evaluation scenarios presented in this chapter, the use case context is considered static: it does not change, as the focus is on the varying situational context such as networking conditions and device resource usage.

The DIVIDE query that is registered to the DIVIDE engine in the evaluation scenarios can be instantiated to an RSP query that calculates the patient's activity index. Given the combination of ontology, use case context and DIVIDE query, this instantiation will happen for the patient described in the use case context. The resulting RSP query will thus be deployed for the DIVIDE component corresponding to this patient's service flat.

Addendum 5.C provides the semantic details of the evaluation use case. It presents relevant definitions from the designed ontology module, an overview of

important triples in the use case context, and details of the internal representation of the discussed DIVIDE query.

5.6.1.3 Realistic dataset for simulation

The evaluations in this chapter are performed using simulations of IoT sensor data in a smart home environment. To ensure that the evaluations are representative, a real-world dataset is employed for these simulations. This dataset is the result of a large scale data collection process in the imec-UGent HomeLab. The HomeLab is a standalone house that can be used as a unique residential testing environment for homecare monitoring use cases. It is equipped with different sensors that measure localization, environmental conditions, user actions and much more. During the data collection process, patients were equipped with an Empatica E4 wearable [49]. This device has a 3-axis accelerometer with a frequency of 32 Hz, as well as multiple other physiological sensors.

For the evaluation scenarios of this work, an anonymous representative part is extracted from the data collected from a random patient. This simulation dataset is left unchanged, except for shifting the observation timestamps to real-time timestamps. In total, it contains an average of 186 observations per second. In fact, for the presented homecare monitoring task, the availability of wearable acceleration data is the only requirement. Nevertheless, by using the realistic dataset for simulation, the volume and variety of the simulated raw sensor data is representative for a real-world service flat.

5.6.2 Compared set-ups

The evaluation scenarios are evaluated on different technical set-ups.

1. **DIVIDE Monitoring set-up:** This is the baseline set-up that deploys all subcomponents of DIVIDE, according to the architecture presented in Section 5.3.1. The set-up uses our implementation presented in Section 5.4.

2. **DIVIDE Local set-up:** This set-up considers the architecture presented in Section 5.3.1, but without the monitoring subcomponents of DIVIDE. This means that the set-up includes the Semantic Mapper and Local RSP Engine on the local device, and the Central Processing Component, Knowledge Base and DIVIDE Core components on the central device. For DIVIDE Core, our implementation of this component as discussed in Section 5.4 is used. However, its task to keep the DIVIDE Meta Model up-to-date is deactivated, as this set-up does not include the Meta Model. Since no subcomponents of the DIVIDE Monitor are deployed, the RSP queries derived by DIVIDE Core always remain active on the Local RSP Engine after DIVIDE Core has registered them to it.

3. **DIVIDE Central set-up:** This set-up is identical to the DIVIDE Local set-up, except for one change: all RSP queries derived by DIVIDE Core are always registered to the Central RSP Engine. Hence, the Local RSP Engine only forwards the monitoring observations and does not evaluate any RSP queries. This is implemented by programmatically issuing a query location update task to the Central RSP Engine for all derived queries, after they are initially registered to the Local RSP Engine.

All implementations of the RSP engine components use the C-SPARQL RSP engine, with the RSP engine wrapper discussed in Section 5.4. The implementation of the Central Reasoner is mocked, since it has no actual task in the evaluation scenarios, apart from exposing an API endpoint for its data stream to which the outputs of the RSP queries can be forwarded.

It is important to note that all evaluation set-ups include the DIVIDE Core subcomponent. This decision is made to ensure that the evaluation investigates the benefits gained by deploying the monitoring subcomponents of DIVIDE. Analyzing the advantages of using DIVIDE Core over other set-ups that do not include DIVIDE, has already been done in our previous work and is therefore considered out of scope for this work [6].

5.6.3 Evaluation scenarios

Two evaluation scenarios are designed for the described homecare monitoring use case. They are discussed in the following subsections.

5.6.3.1 Evaluation scenario 1: updating the RSP query window parameters based on RSP monitoring

The first evaluation scenario focuses on modifying the window parameters of the deployed RSP query that monitors the patient's activity index. The goal of the scenario is to demonstrate how DIVIDE allows dynamically adapting the query window parameters to external factors that prevent the healthcare monitoring from running smoothly.

The focus in this evaluation scenario is on the components that run on the local devices in the network. Hence, for simplicity, only a single DIVIDE component is registered to DIVIDE Central.

Scenario timeline The evaluation scenario takes 5 minutes. In the beginning of the scenario, the nurse call system has no active processes on the local device. After on average 60 seconds into the scenario, a resource-intensive process is started by the nurse call system. 30 seconds later, it starts three more processes. Together, the nurse call processes consume at most 450 MB of RAM, and are very CPU-intensive. These processes are simulated with the Unix workload generator tool *stress*. All started processes continue running for the remainder of the scenario.

This scenario assumes that the networking conditions are too bad to forward the raw accelerometer data to the central servers of the alarm center. This remains the case during the full scenario timeline. Hence, the RSP query is always deployed on the Local RSP Engine and cannot be moved centrally.

Data simulation To simulate the IoT data for the evaluation scenario, a 5-minute chunk of realistic IoT sensor data is extracted from the real-world dataset described in Section 5.6.1.3. During every evaluation run, this chunk is replayed in real-time by a data simulation component. This component is running on an external device that is connected to the considered local device via a local network. This way, the simulation component realistically represents the different IoT sensor gateways. During an evaluation run, the simulation component opens a client connection to the WebSocket server exposed by the wrapper of the Local RSP Engine. Every second, it creates a single-message batch containing all sensor observations of that second, triggers the Semantic Mapper to semantically annotate the messages in the batch, and sends the batch over the WebSocket to the semantic data stream that is registered to the Local RSP Engine. The RDF triple language used in the simulation is N-Triples.

Set-ups The presented scenario is evaluated on the DIVIDE Monitoring set-up and the DIVIDE Local set-up presented in Section 5.6.2. The DIVIDE Central set-up is not considered as the activity index RSP query will always be registered on the Local RSP Engine, also in the DIVIDE Monitoring set-up. For every set-up, the DIVIDE query to calculate the patient's activity index is registered to DIVIDE Core. The static window parameters of the DIVIDE query define a window size of 80 seconds and a query sliding step of 10 seconds.

Global Monitor query In the DIVIDE Monitoring set-up, one Global Monitor query is defined by the end user to be evaluated on the Global Monitor Reasoning Service. This query is presented in Listing 5.6. It monitors the execution time of all RSP queries deployed on the Local RSP Engine of a DIVIDE component. It checks whether the maximum processing time of an RSP query exceeds its sliding step, which would mean that the previous query execution is still ongoing when the next execution needs to start. If this happens, the Global Monitor query defines a window parameter update task for the DIVIDE query corresponding to this RSP query. Both window parameters are updated: the query sliding step is doubled, and the window size is halved. Note that the usage of `GROUP BY` and the aggregations in the Global Monitor query ensure that only one task is outputted for every combination of DIVIDE query and DIVIDE component.

Measurements For every run of the evaluation scenario, the measurements performed by the individual monitors of the Local Monitor are saved. This includes the

Listing 5.6: Global Monitor query deployed on the DIVIDE Monitoring set-up, in evaluation scenario 1 that updates the RSP query window parameters based on the monitored RSP query processing time

```
CONSTRUCT {
    [ a divide-core:DivideWindowParameterUpdateTask ;
      divide-core:isTaskForDivideQueryName ?divideQueryName ;
      divide-core:isTaskForComponentId ?componentId ;
      divide-core:hasUpdatedQuerySlidingStepInSeconds ?maxUpdatedSlidingStep ;
      divide-core:hasUpdatedWindowSizeInSeconds ?minUpdatedWindowSize ]
}
WHERE {
    { SELECT ?componentId ?divideQueryName
             (MAX(?updatedSlidingStep) as ?maxUpdatedSlidingStep)
             (MIN(?updatedWindowSize) AS ?minUpdatedWindowSize)
      WHERE {
          ?component a divide-core:DivideComponent ;
                     divide-core:hasID ?componentId ;
                     divide-core:hasLocalRspEngine ?rspEngine .
          ?rspEngine divide-core:hasRegisteredQuery ?rspQuery .
          ?rspQuery divide-core:hasCorrespondingDivideQuery ?divideQuery ;
                    divide-core:hasAssociatedComponent ?component ;
                    divide-core:hasQuerySlidingStepInSeconds ?querySlidingStep ;
                    divide-core:hasWindowSizeInSeconds ?windowSize .
          ?divideQuery divide-core:hasName ?divideQueryName .

          ?measurement a saref-core:Measurement ;
                       saref-core:hasValue ?maxProcessingTime ;
                       om:hasAggregateFunction om:maximum ;
                       saref-core:isMeasuredIn om:second-Time ;
                       saref-core:relatesToProperty [ a monitoring:RspQueryProcessingTime ] ;
                       saref-core:isMeasurementOf ?rspQuery .

          FILTER (xsd:float(?querySlidingStep) < xsd:float(?maxProcessingTime))

          BIND(xsd:integer(?querySlidingStep) * xsd:float(2)
               AS ?updatedSlidingStep)
          BIND(xsd:integer(FLOOR(xsd:integer(?windowSize) / xsd:float(2)))
               AS ?updatedWindowSize)
      }
      GROUP BY ?componentId ?divideQueryName }
}
```

CPU and RAM usage of the local device, and the processing time on the Local RSP Engine of the deployed RSP query that measures the patient's activity index. For the DIVIDE Local set-up, the Python script of the Device Monitor is manually run so that the CPU and RAM usage is also measured. Moreover, the number of observations in the data window upon every RSP query execution is collected as well.

Technical specifications This evaluation scenario is executed on physical IoT devices. The central device hosting DIVIDE Central and the Central Processing Component is an Intel NUC, model D54250WYKH. It has a 1300 MHz dual-core Intel Core i5-4250U CPU (turbo frequency 2600 MHz) and 8 GB DDR3-1600 RAM. The local device with the Local RSP Engine and the DIVIDE Local Monitor is a Raspberry Pi 3, Model B. This Raspberry Pi model has a Quad Core 1.2 GHz Broadcom BCM2837 64bit CPU, 1 GB RAM and MicroSD storage.

5.6.3.2 Evaluation scenario 2: updating the RSP query location based on network monitoring

The goal of the second evaluation scenario is to evaluate how DIVIDE is able to optimally distribute the RSP queries across the IoT network based on real-time networking conditions. To this end, the scenario focuses on updating the location of the RSP query that monitors the patient's activity index.

This evaluation scenario focuses on both the local and central devices in the network. To this end, it considers an IoT network with a single central device and three local devices. Every local device contains one DIVIDE component that is registered to DIVIDE Central, and represents a single service flat. In the scenario timeline and evaluation measurements, only the interaction between the central device and a single DIVIDE component is considered.

Scenario timeline The duration of this evaluation scenario is 12 minutes. Throughout the full scenario, the properties of the network interface that connects the considered local device with the central device are constantly varied, to simulate varying networking capacity. This simulation adaptively alternates periods with normal (baseline) and worse networking capacity conditions. More specifically, worse networking conditions apply at peak level during the following time periods of the scenario (approximately): minute 1 to 3, minute 5 to 7, and minute 9 to 11. Compared to the baseline capacity, the second period imposes the highest amount of capacity restrictions, while the third period imposes the smallest amount of capacity restrictions.

For simplicity purposes, the networking conditions for the other two DIVIDE components are not varied. Hence, baseline capacity conditions apply. As a consequence, throughout the full scenario, the RSP queries monitoring the activity index for the patients associated to these other two DIVIDE components, are evaluated on the Central RSP Engine.

Data simulation During the runs of this evaluation scenario, the IoT data is simulated in an identical way to the scenario discussed in Section 5.6.3.1. This scenario however uses a 12-minute chunk of IoT from the simulation dataset. The simulation is performed for all three DIVIDE components.

Set-ups This scenario is evaluated on all three set-ups presented in Section 5.6.2: DIVIDE Monitoring, DIVIDE Local and DIVIDE Central. At the start of the simulation for the DIVIDE Monitoring set-up, the RSP query that monitors the patient's activity index will be deployed on the Central RSP Engine. For every set-up, the DIVIDE query to calculate the patient's activity index is registered to DIVIDE Core. The static window parameters of the DIVIDE query define a window size of 60 seconds and a query sliding step of 10 seconds.

Global Monitor query In the DIVIDE Monitoring set-up, the configuration of the DIVIDE Global Monitor contains two Global Monitor queries defined by the end user. The first Global Monitor query is shown in Listing 5.7. It monitors the average network RTT for the connection between every local device and the central device in the IoT network. If this RTT exceeds the threshold of 2 seconds on a device for which queries of the corresponding DIVIDE component are running on the Central RSP Engine, a query location update task is issued that moves this query to the Local RSP Engine. The second Global Monitor query monitors the reverse situation: it ensures that RSP queries deployed on the Local RSP Engine are moved back to the Central RSP Engine whenever the average RTT returns back to 1.6 seconds or lower.

Measurements During the runs of the evaluation scenario, three time durations are measured for every evaluation of the RSP query that measures the patient's activity index: the local processing time, the networking overhead, and the central processing time. The definition of those metrics depends on whether the RSP query is running on the Local RSP Engine or on the Central RSP Engine.

- Query running on the Local RSP Engine:
  - Local processing time: time between the data window trigger of the query evaluation on the Local RSP Engine, *and* the Local RSP Engine sending the generated query result to the Central Reasoner
  - Networking overhead: time between sending the query result by the Local RSP Engine, *and* receiving this result on the Central Reasoner
  - Central processing time: value is 0, since the query is running locally
- Query running on the Central RSP Engine:
  - Local processing time: value is 0, since the query is running centrally

Listing 5.7: Global Monitor query deployed on the DIVIDE Monitoring set-up, in evaluation scenario 2 that updates the RSP query location based on the monitored network RTT

```
CONSTRUCT {
    [ a divide-core:DivideQueryLocationUpdateTask ;
      divide-core:isTaskForDivideQueryName ?divideQueryName ;
      divide-core:isTaskForComponentId ?componentId ;
      divide-core:hasUpdatedQueryLocation [ a divide-core:LocalLocation ] ]
}
WHERE {
    { SELECT DISTINCT ?componentId ?divideQueryName
      WHERE {
          ?device a saref-core:Device ;
                  divide-core:hosts ?component .
          ?component a divide-core:DivideComponent ;
                     divide-core:hasID ?componentId  ;
                     divide-core:hasCentralRspEngine ?rspEngine .
          ?rspEngine divide-core:hasRegisteredQuery ?rspQuery .
          ?rspQuery divide-core:hasCorrespondingDivideQuery ?divideQuery ;
                    divide-core:hasAssociatedComponent ?component .
          ?divideQuery divide-core:hasName ?divideQueryName ;
                       divide-core:hasQueryDeployment [
                           saref-core:isAbout ?component ;
                           divide-core:hasQueryLocation [ a divide-core:CentralLocation ] ] .

          ?measurement a saref-core:Measurement ;
                       saref-core:hasValue ?avgRtt ;
                       om:hasAggregateFunction om:average ;
                       saref-core:isMeasuredIn om:second-Time ;
                       saref-core:relatesToProperty [ a monitoring:RoundTripTime ] ;
                       saref-core:isMeasurementOf ?device .

          FILTER (xsd:float(?avgRtt) >= xsd:float(2.0))
      } }
}
```

- Networking overhead: time between receiving a 1-second sensor data batch on the Local RSP Engine, *and* receiving this batch on the Central RSP Engine (forwarded by the Local RSP Engine), averaged over all batches that are included in the triggered data window of the query evaluation
- Central processing time: time between the data window trigger of the query evaluation on the Central RSP Engine, *and* the Central Reasoner receiving the generated query result

Furthermore, the network RTTs measured by the Network Monitor of the Local Monitor are saved during the evaluation runs. To also measure these RTTs for the DIVIDE Local and DIVIDE Central set-ups, the Python script of the Network Monitor is manually run for those set-ups. Finally, the evaluation measures the number of triples that are sent over the network by the Local RSP Engine in every outgoing network event.

Technical specifications To properly simulate a networking context, this evaluation scenario is run on virtual devices using the in-house iLab.t Virtual Wall environment [50]. All devices are virtual nodes with two 2.40 GHz hexacore Intel Xeon E5645 CPUs and 24 GB DDR3 1333 MHz RAM. For the three local devices, the RAM available to the processes of the device components is limited to 1 GB using Linux Control Groups (Cgroups). All devices are connected via a local area network of which the link characteristics are adaptively configured based on the scenario timeline.

5.6.3.3 Performance evaluation of the DIVIDE Local Monitor and DIVIDE Central

In the two presented evaluation scenarios, the general performance of the monitoring subcomponents of DIVIDE is also measured, in addition to the specific measurements relevant to each evaluation scenario.

Performance evaluation of the DIVIDE Local Monitor For the DIVIDE Local Monitor, the measured performance metrics are the CPU & RAM usage, the number of triples in the output of the Local Monitor aggregation query that is sent over the network, and the processing times of this aggregation query. Similarly to the processing time measured by the RSP Engine Monitor, this metric is defined as the time between the query's data window trigger and the generation of the query results. These performance metrics are all measured during the runs of the first evaluation scenario presented in Section 5.6.3.1.

Performance evaluation of DIVIDE Central Related to the DIVIDE Global Monitor of DIVIDE Central, two performance metrics are calculated.

First, the event processing times of the Global Monitor Reasoning Service are measured. This event processing time is defined as the sum of three parts:

(i) the time from the data window trigger of the Global Monitor query executions, until the data window with all aggregated observations from all Local Monitor instances is added to the DIVIDE Meta Model *and* OWL 2 RL reasoning on the DIVIDE Meta Model is performed;

(ii) the actual execution times of all Global Monitor queries, which are evaluated on the DIVIDE Meta Model;

(iii) and the time from ending the execution of the final Global Monitor query, until the data window with all aggregated observations is removed from the DIVIDE Meta Model *and* OWL 2 RL reasoning is again performed.

Second, the duration is measured of the two tasks that can be issued by the DIVIDE Global Monitor: a query location update task and a window parameter update task. The start of this duration is defined as the data window trigger of the Global Monitor query execution that leads to this task. The end is defined as the time at which the registration of the corresponding RSP engines and their observers at the respective RSP engines is completed by DIVIDE Core.

All performance metrics are measured during the runs of the first evaluation scenario presented in Section 5.6.3.1. The only exception is the duration of the query location update tasks, which is measured during the runs of the second evaluation scenario described in Section 5.6.3.2.

5.7 Evaluation Results

This section presents the results of the evaluations described in Section 5.6. All results contain data of multiple evaluation runs. More details about how the results are aggregated for the evaluations on the DIVIDE Monitoring set-up are presented in Addendum 5.D.

5.7.1 Evaluation scenario 1: updating the RSP query window parameters based on RSP monitoring

Figure 5.4 shows the results of the first evaluation scenario, which is discussed in Section 5.6.3.1. The results show the evolution over time of the processing time of the RSP query that monitors the patient's activity index, together with the real-time RAM and CPU usage of the local device. In addition, the value of the query window parameters is shown. Moreover, Figure 5.5 shows the number of observations in the data window for every evaluation of the considered RSP query. The measurements on the graphs of both figures are averaged over the evaluation runs in time and value.

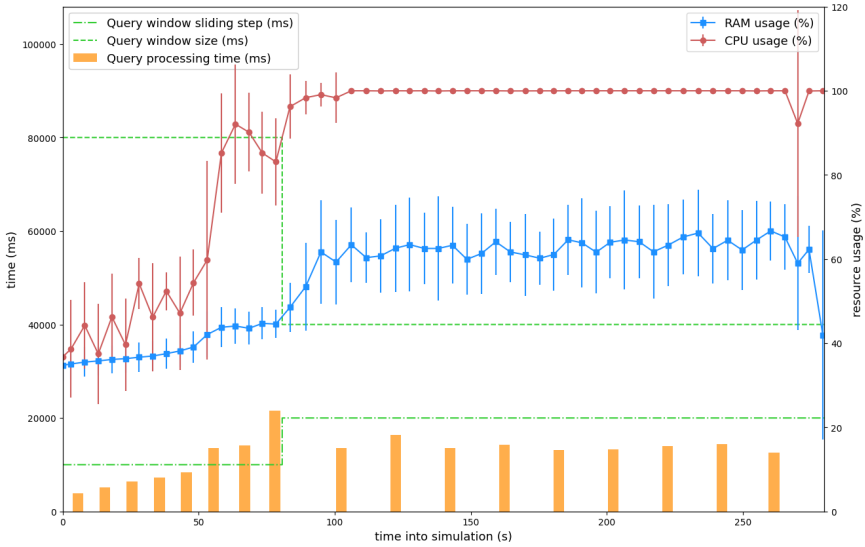
For the DIVIDE Monitoring set-up, Figure 5.4 shows that the query window parameters are updated on average 25 seconds after the first RSP query processing time exceeds the query sliding step of 10 seconds: the query sliding step is doubled and the query window size is halved. After the adaptations, the query is correctly executed every 20 seconds, and the average query processing time remains well below this current sliding step of 20 seconds. Hence, no further adaptations to the window parameters are issued. Moreover, Figure 5.5 proves that the queries are executed on the expected number of observations, also after the window parameters are updated: this value should be around 7,440 (on average 186 observations per second on a window of 40 seconds).

Throughout the scenario, the RAM and CPU usage increase. After on average 58 seconds, the CPU usage shows its largest increase to more than 85% on average. This increase is approximately at the same time at which the first nurse system process is started on the device. After more than 100 seconds into the scenario, the CPU usage reaches and almost constantly remains 100%.

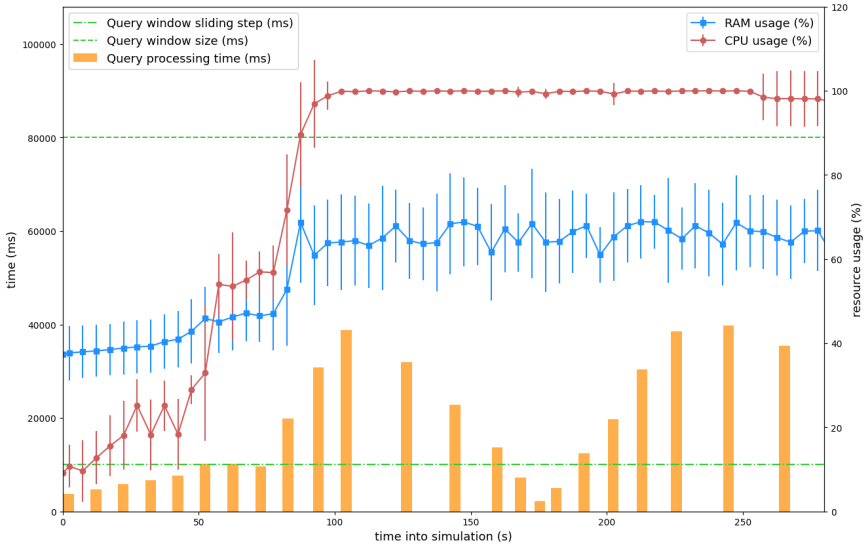
For the DIVIDE Local set-up, a similar trend in the resource usage can be observed on Figure 5.4. However, as the monitoring subcomponents of DIVIDE are not deployed, no adaptations to the window parameters are made. As a consequence, after more than 100 seconds into the evaluation, the RSP query is no longer correctly executed every 10 seconds. Often, the period between successive query executions is larger (at most on average more than 22 seconds) or smaller (on average less than 7 seconds at the smallest), causing an irregular pattern. Moreover, the query processing times are irregular as well: they are larger than 30 seconds on average on seven executions, with an average maximum value of more than 39 seconds. Figure 5.5 demonstrates that the number of observations in the data window on which the RSP query is executed, is also way lower than expected. The value should be around 14,880 (on average 186 observations per second on a window of 80 seconds), but is often way lower. This means that the set-up cannot keep up with the data velocity, and is thus ignoring many sensor observations in most query evaluations.

5.7.2 Evaluation scenario 2: updating the RSP query location based on network monitoring

Figure 5.6 shows the results of the second evaluation scenario, which is discussed in Section 5.6.3.2. The graph visualizes the evolution over time of the local processing time, network overhead and central processing time related to the evaluation of the RSP query that monitors the patient's activity index. Moreover, the network RTT between the local and central device, and the number of triples sent over the network in every outgoing event is plotted as well. All measurements are averaged over the evaluation runs in time and value.



(a) DIVIDE Monitoring set-up



(b) DIVIDE Local set-up

Figure 5.4: Part 1 of the results of evaluation scenario 1 that updates the RSP query window parameters based on the monitored query processing time. The processing times are shown for every evaluation of the RSP query, on the timestamp of the data window trigger of the query execution. Moreover, the RAM & CPU usage of the local device is shown. All results are averaged on both axes over the evaluation runs. The error bars represent standard deviations.

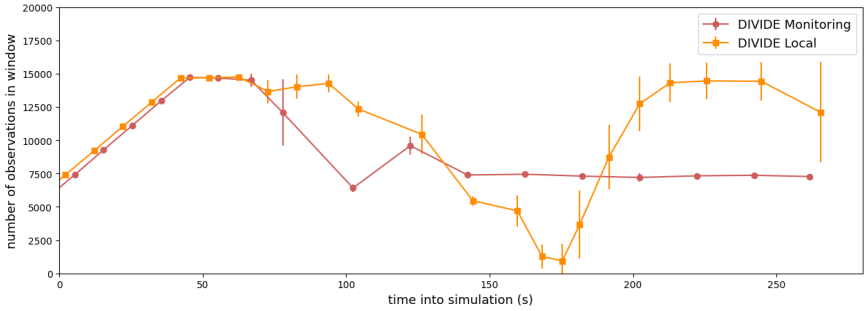


Figure 5.5: Part 2 of the results of evaluation scenario 1 that updates the RSP query window parameters based on the monitored query processing time. The graph plots the number of observations in the data window of every evaluation of the RSP query, on the timestamp of the data window trigger of the query execution. All results are averaged on both axes over the evaluation runs. The error bars represent standard deviations.

For the DIVIDE Monitoring set-up, the RSP query is moved between the Local RSP Engine and Central RSP Engine four times. It is moved from the central to the local device during two of the three simulated periods of bad networking conditions, and moved back to the central device after the network conditions have improved again. The delay between the start or end of the period of bad networking conditions and the actual query move varies between approximately 50 and 80 seconds on average. This delay is caused by the Global Monitor queries, which require the average RTT to be above or below a certain threshold. This average is calculated by the Local Monitor RSP Engine on a data window of 60 seconds. Therefore, queries are only moved if bad or good networking conditions persist for a certain period. The network overhead of the RSP query processing varies with a similar pattern as the measured network RTT. Concretely, it varies between 109 ms and 10,412 ms on average, with an average value of 2,339 ms (SD 1,993 ms). The local processing times and central processing times vary less: they are on average 232 ms (SD 330 ms) and 370 ms (SD 302 ms), respectively, with respective maximums of on average 840 ms and 771 ms.

Considering the query processing times for the DIVIDE Local and DIVIDE Central set-up, the average values for those metrics are more constant than in the DIVIDE Monitoring set-up. Concretely, the average local processing time for the DIVIDE Local set-up is 716 ms (SD 77 ms), while the average central processing time for the DIVIDE Central set-up is 647 ms (SD 73 ms). Similarly to the DIVIDE Monitoring set-up, the network overhead largely follows the same pattern as the measured RTT in both set-ups.

Focusing on the number of triples sent over the network in messages by the Local RSP Engine, big differences can be observed between the different set-ups. For the DIVIDE Monitoring set-up, this number is on average 443,656 triples in total over

the full scenario. For the DIVIDE Local and DIVIDE Central set-ups, this average sum of triples is 355 and 658,775, respectively.

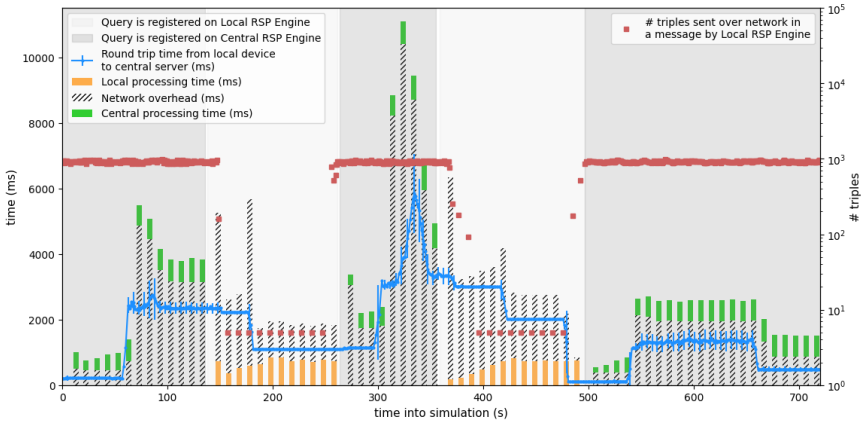
Figure 5.7 shows additional results of the second evaluation scenario, specifically for the DIVIDE Monitoring set-up. For these results, there is only one change to the evaluation set-up, compared to the set-up presented in Section 5.6.3.2 of which the results are shown in Figure 5.6. This change is the value of the threshold for the network RTT in the Global Monitor queries deployed on the DIVIDE Global Monitor. In the original scenario, the RSP query is moved to the Local RSP Engine when the RTT is higher than the threshold of 2 seconds, and it is moved back to the Central RSP Engine when the RTT is lower than the threshold of 1.6 seconds.

- Figure 5.7a shows the results of changing these thresholds to 1 and 0.8 seconds, respectively. With this change, the location of the RSP queries changes four times. In the first period with bad networking conditions, the query is moved to the Local RSP Engine. Thereafter, the query does not move back to the Central RSP Engine during the period with better network conditions, because the average RTT does not get below the lower threshold of 0.8 seconds. Only after the second period of bad networking conditions is finished, the RSP query is moved back to the Central RSP Engine. Finally, the query is moved one more time to the Local RSP Engine during the third period of bad networking conditions, and back to the Central RSP Engine after this period. In this set-up, on average 217,713 triples are sent over the network by the Local RSP Engine.

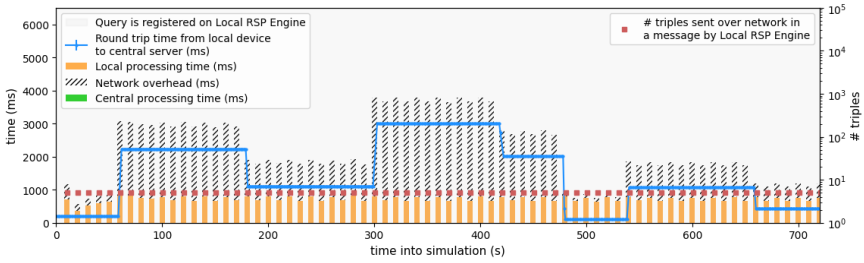
- Figure 5.7b shows the results of changing these thresholds to 3 and 2.4 seconds, respectively. With this configuration, the RSP query is only moved once to the Local RSP Engine and back, caused by the second period of bad networking conditions that poses the largest restriction on the network capacity. The average total number of triples sent over the network in this set-up is 564,142.

5.7.3 Performance evaluation of the DIVIDE Local Monitor and DIVIDE Central

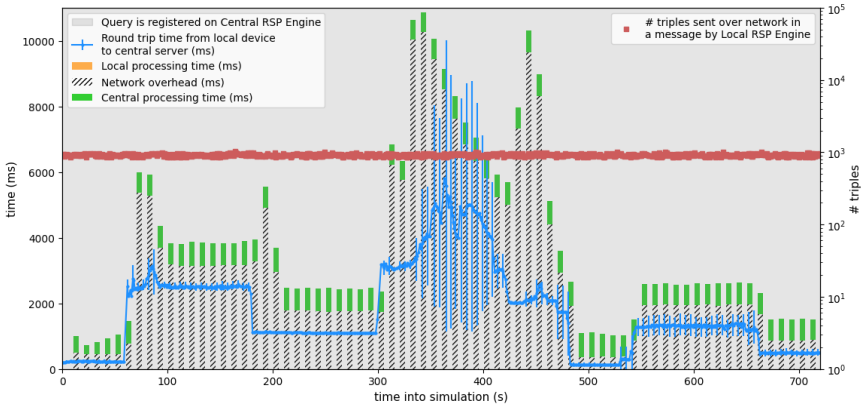
Figure 5.8 shows the results of the performance evaluation of the DIVIDE Local Monitor, measured on evaluation scenario 1. This figure shows that the CPU usage of the Local Monitor is 10% or lower for on average 72% of the measurements throughout the evaluation runs. Only in on average 1% of the measurements, the CPU usage exceeds 30%. Moreover, the average RAM used by the Local Monitor is 100 MB (SD 2 MB). The execution time of the Local Monitor aggregation query is 1,022 ms on average (SD 349 ms). The average execution time increases after 60 seconds when the nurse call system processes are started in the evaluation scenario.



(a) DIVIDE Monitoring set-up

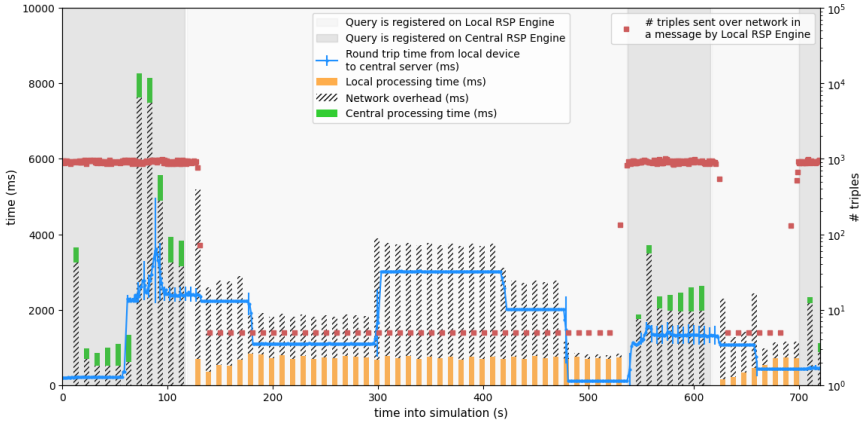


(b) DIVIDE Local set-up

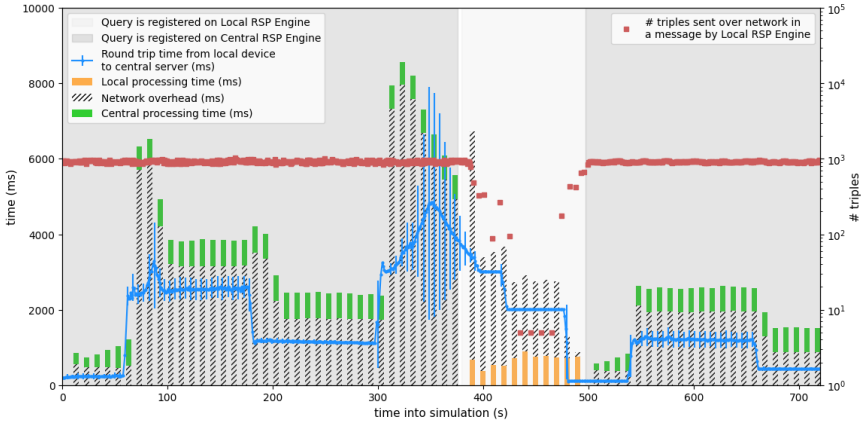


(c) DIVIDE Central set-up

Figure 5.6: Results of evaluation scenario 2 that updates the RSP query location based on the monitored network RTT. The time duration measurements are shown for every RSP query evaluation, on the timestamp of the data window trigger of the query execution. Moreover, the graphs plot the network RTT and number of triples in outgoing network events. All results are averaged on both axes over the evaluation runs. The error bars represent standard deviations.

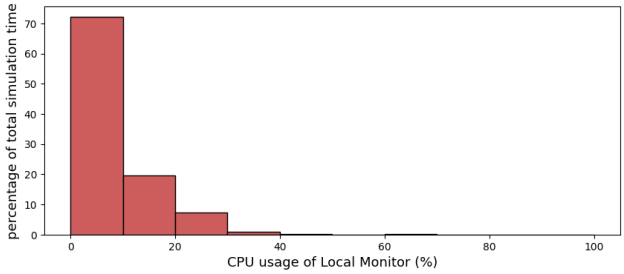


(a) Lower RTT threshold in Global Monitor queries: move to central if value is 1 second or higher, move to local if value is 0.8 seconds or lower

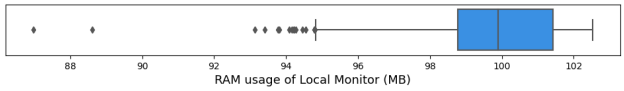


(b) Higher RTT threshold in Global Monitor queries: move to central if value is 3 seconds or higher, move to local if value is 2.4 seconds or lower

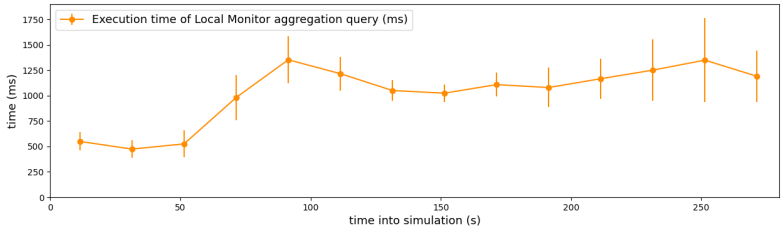
Figure 5.7: Additional results of evaluation scenario 2 that updates the RSP query location based on the monitored network RTT. These results are only shown for the DIVIDE Monitoring set-up. The only change to the evaluation set-up compared to the results shown in Figure 5.6 are the thresholds for the RTT in the Global Monitor queries that decide when the RSP query should be moved between the Local RSP Engine and Central RSP Engine. The time duration measurements are shown for every RSP query evaluation, on the timestamp of the data window trigger of the query execution. Moreover, the graphs plot the network RTT and number of triples in outgoing network events. All results are averaged on both axes over the evaluation runs. The error bars represent standard deviations.



(a) Histogram with distribution of Local Monitor CPU usage over the full scenario and multiple runs



(b) Distribution of Local Monitor memory (RAM) usage over the full scenario and multiple runs



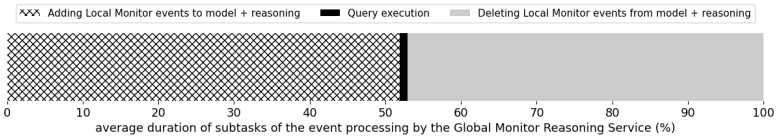
(c) Timeline of execution time of Local Monitor aggregation query, averaged over multiple runs of the scenario (error bars represent standard deviations)

Figure 5.8: Results of the performance evaluation of the DIVIDE Local Monitor, measured on evaluation scenario 1

Figure 5.9 shows the results of the performance evaluation of the DIVIDE Global Monitor Reasoning Service. It shows the distribution of the event processing time, which is on average 1,066 ms (SD 316 ms). The distribution of this time over the three parts is also shown: on average 52% of the time is spent on adding the Local Monitor events to the DIVIDE Meta Model and performing reasoning, only 1% is spent on average on the execution of the Global Monitor queries, and the remaining 47% is spent on removing the events from the DIVIDE Meta Model and performing reasoning again.

Finally, Figure 5.10 presents the distribution of the duration of the tasks that can be issued by the DIVIDE Global Monitor, over multiple runs. For a query location update task, moving a query to the Local RSP Engine and the Central RSP Engine takes on average 18,254 ms (SD 1,661) and 6,395 ms (SD 3,383 ms), respectively. For a window parameter update task, this value is on average 7,093 ms (SD 2,134 ms). Specifically for this task, the part of the task duration spent on the performed

(a) Distribution of the event processing times over the full scenario and multiple runs



(b) Average duration of subtasks of the event processing times, averaged over the full scenario and multiple runs

Figure 5.9: Results of the performance evaluation of the Reasoning Service of the DIVIDE Global Monitor, measured on evaluation scenario 1

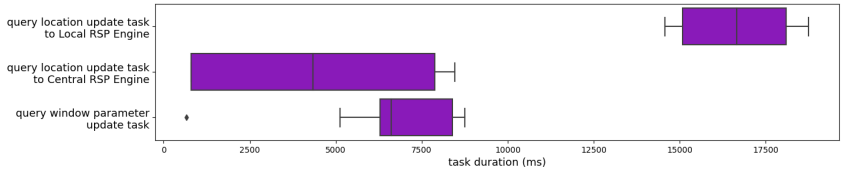


Figure 5.10: Distribution of the duration of the tasks that can be issued by the DIVIDE Global Monitor, over multiple runs. The duration of the window parameter update task is measured in the runs of evaluation scenario 1, the duration of the query location updates in the runs of evaluation scenario 2.

window parameter substitution of the query derivation (step 5 as explained in Section 5.2.2) is 184 ms (SD 28 ms).

5.8 Discussion

DIVIDE is a semantic component that can be deployed in a semantic IoT platform to derive and manage the relevant queries for the stream processing components of the platform in an automatic, adaptive and context-aware way. In our previous work on DIVIDE, we have already demonstrated the added value of using DIVIDE Core over other state-of-the-art set-ups that involve real-time semantic reasoning on IoT data streams [6]. Hence, the focus of this chapter and its evaluations is on the improved methodological design of DIVIDE, involving the different central and local subcomponents that allow performing situational context monitoring to manage the configuration and distribution of the stream processing queries across the network. Therefore, the different evaluations performed in this chapter compare a set-up involving all subcomponents of DIVIDE (DIVIDE Monitoring) with two set-ups that only involve DIVIDE Core and do not perform any situational context monitoring (DIVIDE Central and DIVIDE Local).

The first evaluation scenario focuses on adapting the window parameter configuration of queries, based on the performance of the query evaluation at the Local RSP Engine. The local device in the evaluation is a Raspberry Pi, which is a typical IoT device with few resources. The evaluation results in Figure 5.4 and 5.5 show that the DIVIDE Monitoring set-up can intelligently update the query window parameters, according to the end user definitions in the Global Monitor queries. Moreover, the results show that by lowering the data window size and execution frequency (i.e., increasing the sliding step) whenever the device does not have enough resources to keep up with the data, DIVIDE helps ensuring that the query remains running correctly in terms of frequency and data in its input window. In comparison, the DIVIDE Local set-up cannot dynamically alter the window parameters, and thus cannot keep up with the data volume and velocity, as soon as the resource usage on the Raspberry Pi starts increasing because of the nurse call system processes. This might lead to ignored data and incorrect, delayed data processing, as shown in the evaluation results. In the worst case, it could even lead to crashes of the Local RSP Engine.

Looking at the Global Monitor query that updates the window parameters, it should be noted that it currently does not prevent the given DIVIDE task from being issued multiple times for the same DIVIDE component and DIVIDE query. This could lead to an undesirable configuration of the window parameters where the window size is smaller than the sliding step, causing data to be ignored in the calculation of the activity index. Hence, depending on the use case, the Global Monitor query could be altered to avoid this. Nevertheless, it is important to realize that the Global Monitor queries can avoid the system from crashing. Hence, depending on the use case, it might be allowed to ignore some data to prevent the system from crashing and thus processing no data at all. Moreover, although this was assumed to be impossible in the evaluated scenario, note that moving the RSP query to the central device might also be a valid solution if the local query performance keeps on decreasing. Hence, in ideal circumstances, multiple Global Monitor queries are deployed that jointly consider RSP performance and networking conditions.

The second evaluation scenario focuses on updating the query location in the network based on the monitored network properties. By varying the properties of the relevant network links, a realistically varying networking capacity has been simulated. The results in Figure 5.6 demonstrate the differences between the set-ups. In the DIVIDE Local set-up, the networking overhead is the smallest off all set-ups throughout the scenario, since only the query results should be forwarded to the central device. The local processing time is also rather constant and well below 1 second. However, in this particular use case, this set-up is not preferred. This is because the alarm center wants to have as much raw data on the central servers as possible, to ensure patient security. The alternative set-up without the monitoring subcomponents of DIVIDE is DIVIDE Central. As shown in the results, this set-up is not ideal either, for multiple reasons. First, since all raw data needs to be forwarded to the

central server, the network is heavily burdened. In some evaluation runs (as shown in Addendum 5.D), this leads to heavily accumulating delays in the data processing, which is problematic when real-time follow-up is required. In the scenario timeline, the bad networking conditions did not take longer than 2 minutes. In contrast, these bad conditions could persist for a much longer period of time in real-world scenarios, posing the risk of the networking overhead and thus processing delays to even further accumulate. A second disadvantage of the DIVIDE Central set-up is the associated server cost. In the evaluation results, the processing times are well below 1 second. However, it should be noted that the evaluation was performed with only three local devices. In a real-world set-up, this number is likely to be at least an order of magnitude larger. This would put much higher requirements on the central server resources to guarantee the same level of performance. In contrast to the DIVIDE Local and DIVIDE Central set-ups, the DIVIDE Monitoring set-up combines the best of both worlds by adaptively changing the query distribution based on the network conditions, taking into account the use case requirements configured by the end users in the Global Monitor queries. Specifically for the considered homecare monitoring use case, this set-up optimally balances the trade-off between patient security and server cost. It does this in a dynamic environment, taking situational context into account when balancing the trade-off: if the network does not allow forwarding raw data even if this is to be preferred, moving the query locally and forwarding only aggregated insights is still better than having no up-to-date, correct insights centrally. In other words, given the current situational context across the IoT platform *which cannot be altered*, DIVIDE allows optimally balancing all use case requirements defined through Global Monitor queries *by altering the query distribution and configuration*.

The evaluation use case of this chapter focuses on the homecare monitoring task of calculating a patient's activity index. This is an aggregation of raw accelerometer data, and thus inherently entails less information. This means that the same activity index value calculated over a data window of 60 seconds could correspond to different activity patterns over the course of that minute. Depending on a patient's medical condition, the priority of knowing this exact activity pattern and thus having access to the raw data differs. Therefore, the second evaluation scenario has been run for different versions of the Global Monitor queries, containing other RTT thresholds for moving the query between the Local and Central RSP Engine. Compared to the original results presented in Figure 5.6, Figure 5.7a and Figure 5.7b show the results for running the evaluation scenario with lower and higher RTT thresholds, respectively. In essence, higher RTT thresholds in these Global Monitor queries correspond to a higher priority of having the raw data centrally and thus running the RSP query centrally. As explained in Section 5.6.1.1, the higher thresholds could correspond to a fall-prone patient, where access to the raw data is the most important and thus only sacrificed when the networking conditions are really bad. On the other

hand, for a dementia patient, lower granularity insights into the patient's activity pattern suffice and thus lower RTT thresholds, that allow moving RSP queries to the local device more quickly, are allowed.

In the networking evaluation, the results about the number of the triples sent over the network in the different set-ups clearly demonstrate the impact of the query location on the network burden. In this scenario, when running the query locally, only 5 triples are sent over the network every 10 seconds. Instead, when running the query centrally, on average 9300 triples are sent over the network in 10 seconds (on average 186 observations per second for 10 seconds, with 5 triples per observation). This demonstrates, for a realistic data volume and velocity, the relevance and importance of DIVIDE allowing to move the RSP query locally when networking conditions get worse. For completeness, it should be noted that the choice of RDF triple language can also impact the evaluation results. By using another language than N-Triples such as RDF/Turtle and optimally exploiting the usage of prefixes, the size of the messages can be lowered to increase network efficiency. However, this would happen at the expense of requiring more central resource-intensive parsing.

Inspecting the results of the performance evaluation of the DIVIDE Local Monitor in Figure 5.8, it is clear that the Local Monitor has an acceptable usage of CPU and RAM resources, even on a low-end device such as a Raspberry Pi. As can be observed, the execution times of the aggregation queries are impacted by the other active processes on the system, but are still below 1.5 seconds on average when the CPU usage is 100%.

The performance evaluation results for the Global Monitor in Figure 5.9 show that the event processing times on the Global Monitor Reasoning Service are just above 1 second on average, with a few outliers. On average 99% of this time is spent on adding the monitoring data to the DIVIDE Meta Model and removing it again. Both steps involve semantic reasoning by Apache Jena. Implementing this Reasoning Service with more efficient state-of-the-art semantic reasoners would allow a significant decrease of the total event processing times, especially if the reasoner supports incremental reasoning. Such reasoners only need to perform semantic reasoning on the updated parts of the data model, instead of having to perform the whole semantic reasoning process again on every update like Apache Jena does.

Inspecting the distribution of the duration of the tasks that can be issued by the DIVIDE Global Monitor in Figure 5.10, it is clear that these tasks take a while. This duration involves the event processing on the Global Monitor Reasoning Service until the output of the Global Monitor query is generated, the parsing by the DIVIDE Monitor translator, and the actual task execution by DIVIDE Core. The query location update tasks mainly involve communication over the network with both RSP engines. The move to the Local RSP Engine is issued in bad networking conditions, which explains the high durations with an average of more than 18 seconds. For the query window parameter update task, the final query derivation steps performed by DIVIDE

Core take only 184 ms on average. Hence, the remaining seconds are also spent on sending the query updates to the Local RSP Engine. The high task duration can thus be explained by taking into account the 100% CPU usage on the local device when the query update requests are received. In addition, all high task durations should be put into perspective. First of all, during the query updates, all received streaming data is buffered by the Local RSP Engine, so that no data is ignored or removed. Moreover, the configuration of the Global Monitor queries ensures that the tasks are issued for a reason: even if there is a single larger gap than desired between two query executions, this is still better than not executing the task and leaving the RSP query configuration and distribution as is. By letting the end user define the Global Monitor queries and thus the thresholds of the monitored situational context properties, one can intelligently tweak when and how often the query configuration and distribution is updated.

The Global Monitor queries are the main tool for end users to configure the behavior of the monitoring subcomponents of DIVIDE. Hence, it is important to make the process of defining those Global Monitor queries as user-friendly as possible. In this chapter, we have made a first attempt in improving the user-friendliness by suggesting a BNF grammar to define the actuation rules for the Global Monitor queries. The goal of this grammar is to hide the inner semantic details of how the monitor measurements, the DIVIDE meta-information and the issued tasks are described with the DIVIDE Meta Model ontology. These semantic details are irrelevant for end users, and require knowledge of Semantic Web technologies such as RDF and SPARQL. Using such a grammar, an end user only needs to know the DIVIDE terminology, what tasks can be issued, and what properties are being monitored. In addition, BNF grammar rules are less verbose than SPARQL queries, increasing the user-friendliness.

To put DIVIDE into production, further steps in improving the user-friendliness should be made. In general, user-friendly interfacing is required to optimally configure the system with all DIVIDE components, DIVIDE queries and Global Monitor queries. Such an interfacing system could also automatically suggest relevant Global Monitor queries based on the configured use case requirements, for example to avoid the RSP engine from crashing or accumulating processing delay in case of high resource usage. An interesting addition would be a priority-based system, where the Global Monitor query conditions and thresholds for updating the query distribution would be dependent on assigned priorities of having central access to the raw sensor data. Such a priority could be assigned based on the window parameters of the RSP queries, as they define the degree of information loss between the raw data and the outputs of the RSP queries: the lower the size of the data window, the smaller the information loss would be if only the aggregated query outputs are available and not the original raw data. Other options could be to assign priorities per DIVIDE query, or even based on the medical conditions in the patient profile in healthcare applications. Furthermore, in some applications, changing RSP query configuration details such as window parameters is not allowed. This is for example the case if the query

outputs are processed by a machine learning algorithm that requires certain input features to be aggregations made on a data window of a specific size. The interfacing system should then allow an end user to define such priorities and conditions in a user-friendly way, and automatically translate them into the correct Global Monitor queries. It should be noted that some of these suggestions would require some small changes to the design of DIVIDE, for example to add some specific parameters or priorities in the semantic descriptions of DIVIDE queries or DIVIDE components. As for the rest, everything is readily available in the methodological design of DIVIDE to make such more user-friendly interfacing possible.

As the design of the subcomponents of DIVIDE is generic and modular, it is possible to easily extend its functionality in the future. Adding new parameters to be monitored to the existing individual monitors only requires a few adaptations. First, the new properties should be added to the `Monitoring` module of the DIVIDE Meta Model ontology. Second, the implementation of the DIVIDE Local Monitor should be updated: the semantic mapper should include the new properties, and the individual monitors should continuously collect measurements for the new properties and output them in the required JSON message format. Due to the modular design, the implementation of the individual monitors can also be easily replaced with another implementation, without requiring further changes. New individual monitors can also be integrated in a similar way. Some relevant new monitoring properties to be included as future work are the energy consumption of the processing devices, and whether or not the network connection between the local and central device is metered. The latter can be especially relevant when extending the architecture to mobile devices, to avoid that high volumes of raw data are being sent over a metered connection. Focusing on the Local Monitor aggregation queries, new aggregations can also be easily added to the existing implementation. The current query already generically aggregates all properties using the DIVIDE Meta Model ontology, such that no changes are required if new properties are being monitored.

Future extensions to the methodological design of DIVIDE could also consist of allowing the Global Monitor to take additional meta-information into account. For example, in a more sophisticated cascading architecture, more details about the data and query deployment might be relevant to be taken into account by the Global Monitor queries. To enable this, several changes are required. First, the `DivideCore` module of the Meta Model ontology needs to be extended. Moreover, the implementation of the integration of the DIVIDE Meta Model needs to manage the corresponding new triples in the Meta Model, and the implementation of DIVIDE Core needs to be altered to keep the new meta-information in the Meta Model up-to-date at all times. Another possible extension to the Global Monitor design could be to perform monitoring on the central device as well. This could include monitoring the performance of the Central RSP Engine, as well as the correct inner workings and network communication of the DIVIDE implementation. In addition, future research could also

look into updating the query distribution task issued by the DIVIDE Global Monitor in a more fine-grained manner, e.g. by including multiple edge devices into the architecture. Related to this, new RSP or reasoning engines could also be deployed on or removed from components in the network, based on the monitored situational context. This would further increase the dynamism of the query distribution. The existing implementation of deploying Local Monitor instances over SSH across the IoT network could be leveraged for this.

To start using DIVIDE in a real-world production environment, several changes to its design and implementation are required. On design level, support should be added to update the window parameters of queries with multiple stream windows, and stream windows that specify the data window interval relative to the current time. This is not integrated into the current design of DIVIDE, as this was not required to demonstrate its capabilities and validate the research questions. On implementation level, several general and specific improvements are required. These improvements were out of scope for this work, as the focus of this research is on demonstrating the possibilities of the monitoring aspect of DIVIDE on a methodological level and validating it with a first implementation. A first specific possible improvement is the implementation of the network monitor. In a production environment, it should monitor multiple networking properties in a more sophisticated way. To achieve this, one could look into using existing network monitoring tools. Second, the implementation of the DIVIDE Monitor Translator could be improved to support multiple window parameters with different variable names. Moreover, the current chronological parsing and forwarding of issued tasks could be replaced by an improved algorithm that intelligently handles conflicting, duplicate and frequently recurring tasks. Third, enabling the RSP Engine Monitor requires integrating the original implementation of this RSP engine with our RSP engine wrapper implementation. This might require small adaptations to the source code of the RSP engine, which does not impose an issue as the source code of most engines is open source. If this is not the case, the required monitoring data could also be extracted from the logs of the RSP engine wrapper.

5.9 Conclusion

This chapter has presented DIVIDE, which is a semantic component that can be deployed in a cascading reasoning architecture of a semantic IoT platform. In the work, the methodological design and a first implementation of DIVIDE is discussed and evaluated on a homecare monitoring use case. Looking back at the research objectives outlined in Section 5.1, we can conclude that we have achieved those in the following ways:

1. The design of the monitoring subcomponents of DIVIDE enables the continuous monitoring of various relevant parameters of the situational context

in which tasks are deployed across the stream processing components in the IoT network. Using the designed DIVIDE Meta Model ontology, the Local Monitor can monitor these parameters through multiple individual monitors: network characteristics with the Network Monitor, resource usage of the stream processing devices with the Device Monitor, and data stream properties and real-time performance of the stream processing components with the RSP Engine Monitor.

2. By forwarding aggregations of the Local Monitor observations to the Global Monitor Reasoning Service and continuously evaluating the configured Global Monitor queries on the DIVIDE Meta Model containing meta-information about the system, actions can be taken based on the monitored situational context. These actions can be defined by DIVIDE tasks specified in the output of the Global Monitor queries. Two tasks are supported in the current design of DIVIDE: updating the window parameter configuration of the deployed RSP queries (i.e., updating the query's window size and/or sliding step), and updating the distribution of those queries by moving them between the Local and Central RSP Engine. The presented evaluation results prove that these tasks can be successfully performed by our implementation of DIVIDE.

3. Through the definition and configuration of use case specific Global Monitor queries, an end user can dynamically configure how the situational context parameters should influence the RSP query configuration and distribution in the network. This way, the actuation can be mapped to the requirements of every individual use case. By suggesting a BNF grammar to define these queries as actuation rules, no end user knowledge of the Meta Model ontology concepts or Semantic Web technologies should be required. The evaluation results presented in this chapter show that the dynamic approach to configuring RSP query window parameters and the RSP query distribution in the network, allows reacting to situational context parameters such as varying device resource usage or networking conditions. The results demonstrate that the usage of a set-up with the monitoring subcomponents of DIVIDE can better deal with use case specific requirements in such dynamic environments, compared to alternative static set-ups. In summary, DIVIDE can increase the percentage of time that an optimal balance of use case specific trade-offs is guaranteed. This way, it automatically achieves more efficient stream processing.

4. Through our design of DIVIDE, we have laid the foundation of monitoring the relevant situational context information on local and edge components, and updating the RSP query configuration and distribution based on this information in an automated way. Laying this foundation is done by generically designing the Meta Model ontology and subcomponents of DIVIDE. Given this generic

and modular design, new properties can be easily monitored by extending the Meta Model ontology and locally extending or adding individual monitors.

5. By building further on the design and implementation of our previous work on DIVIDE [6], the presented design of DIVIDE allows deriving and managing the RSP queries in an adaptive and context-aware way, based on domain knowledge and use case specific context.

Future work presents multiple interesting directions. First, to put DIVIDE into production, the user-friendliness of the Global Monitor query definition should be further improved by designing an interfacing system that intelligently translates use case requirements and assigned priorities into the deployed queries. Second, several design and implementation improvements should be performed to make the system more robust and complete. Possible improvements include improving the monitoring of network characteristics and supporting queries with multiple input data windows. Third, the system design could be extended to allow for dynamic deployment of query engines across the IoT network depending on the monitored situational context.

Funding

This research is part of the imec.ICON project PROTEGO (HBC.2019.2812), co-funded by imec, VLAIO, Televic, Amaron, Z-Plus and ML2Grow. This research is also partly funded by the FWO Project FRACTION (nr. G086822N).

Availability of data and materials

Supportive information relevant to the evaluation set-ups of this chapter is publicly available at https://github.com/IBCNServices/DIVIDE/tree/master/jnsm2023. This page also refers to multiple other publicly available pages. These include the DIVIDE Meta Model ontology at https://github.com/IBCNServices/DIVIDE/tree/master/meta-model, the source code of our first implementation of the different DIVIDE subcomponents at https://github.com/IBCNServices/DIVIDE/tree/master/src, additional details of the DAHCC ontology at https://dahcc.idlab.ugent.be, and the described evaluation dataset used for simulation at https://dahcc.idlab.ugent.be/dataset.html. A dedicated tag of the DIVIDE repository is created at https://github.com/IBCNServices/DIVIDE/releases/tag/jnsm-2023, which represents the version of implementation (`src` folder) and Meta Model ontology (`meta-model` folder) resulting from the current work.

References

[1] X. Su, J. Riekki, J. K. Nurminen, J. Nieminen, and M. Koskimies. *Adding semantics to Internet of Things*. Concurrency and Computation: Practice and Experience, 27(8):1844–1860, 2015. doi:10.1002/cpe.3203.

[2] C. C. Aggarwal, N. Ashish, and A. Sheth. *The Internet of Things: A Survey from the Data-Centric Perspective*. In C. C. Aggarwal, editor, Managing and Mining Sensor Data, pages 383–428. Springer US, 2013. doi:10.1007/978-1-4614-6309-2_12.

[3] I. Kalamaras, N. Kaklanis, K. Votis, and D. Tzovaras. *Towards Big Data Analytics in Large-Scale Federations of Semantically Heterogeneous IoT Platforms*. In L. Iliadis, I. Maglogiannis, and V. Plagianakos, editors, Artificial Intelligence Applications and Innovations: Proceedings of AIAI 2018 IFIP 12.5 International Workshops, pages 13–23, Cham, Switzerland, 2018. Springer. doi:10.1007/978-3-319-92016-0_2.

[4] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. *Semantics for the Internet of Things: Early Progress and Back to the Future*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012. doi:10.4018/jswis.2012010101.

[5] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. *Stream reasoning: A survey and outlook*. Data Science, 1(1-2):59–83, 2017. doi:10.3233/DS-170006.

[6] M. De Brouwer, B. Steenwinckel, Z. Fang, M. Stojchevska, P. Bonte, F. De Turck, S. Van Hoecke, and F. Ongenae. *Context-aware query derivation for IoT data streams with DIVIDE enabling privacy by design*. Semantic Web, 14(5):893–941, 2023. doi:10.3233/SW-223281.

[7] M. De Brouwer, D. Arndt, P. Bonte, F. De Turck, and F. Ongenae. *DIVIDE: Adaptive Context-Aware Query Derivation for IoT Data Streams*. In Joint Proceedings of the International Workshops on Sensors and Actuators on the Web, and Semantic Statistics, co-located with the 18th International Semantic Web Conference (ISWC 2019), volume 2549, pages 1–16, Aachen, 2019. CEUR Workshop Proceedings. Available from: https://ceur-ws.org/Vol-2549/article-01.pdf.

[8] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride. *RDF 1.1 concepts and abstract syntax*. W3C Recommendation, World Wide Web Consortium (W3C), 2014. Available from: https://www.w3.org/TR/rdf11-concepts/.

[9] W3C OWL Working Group. *OWL 2 Web Ontology Language*. W3C Recommendation, World Wide Web Consortium (W3C), 2012. Available from: https://www.w3.org/TR/owl2-overview/.

[10] S. Harris and A. Seaborne. *SPARQL 1.1 Query Language*. W3C Recommendation, World Wide Web Consortium (W3C), 2013. Available from: https://www.w3.org/TR/sparql11-query/.

[11] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. *OWL 2 Web Ontology Language Profiles (Second Edition)*. W3C Recommendation, World Wide Web Consortium (W3C), 2012. Available from: https://www.w3.org/TR/owl2-profiles/.

[12] X. Su, E. Gilman, P. Wetz, J. Riekki, Y. Zuo, and T. Leppänen. *Stream reasoning for the Internet of Things: Challenges and gap analysis*. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), pages 1–10, New York, NY, USA, 2016. Association for Computing Machinery (ACM). doi:10.1145/2912845.2912853.

[13] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. *C-SPARQL: a continuous query language for RDF data streams*. International Journal of Semantic Computing, 4(1):3–25, 2010. doi:10.1142/S1793351X10000936.

[14] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte, and O. Corcho. *RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems*. International Journal on Semantic Web and Information Systems (IJSWIS), 10(4):17–44, 2014. Available from: https://dl.acm.org/doi/10.5555/2795081.2795083.

[15] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. Van Harmelen. *Towards expressive stream reasoning*. In Semantic Challenges in Sensor Networks, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. doi:10.4230/DagSemProc.10042.4.

[16] M. De Brouwer, F. Ongenae, P. Bonte, and F. De Turck. *Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions*. Sensors, 18(10):3514, 2018. doi:10.3390/s18103514.

[17] P. Bonte, R. Tommasini, E. Della Valle, F. De Turck, and F. Ongenae. *Streaming MASSIF: cascading reasoning for efficient processing of iot data streams*. Sensors, 18(11):3832, 2018. doi:10.3390/s18113832.

[18] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler. *N3Logic: A logical framework for the World Wide Web*. Theory and Practice of Logic Programming, 8(3):249–269, 2008. doi:10.1017/S1471068407003213.

[19] L. Daniele, F. den Hartog, and J. Roes. *Created in Close Interaction with the Industry: The Smart Appliances REFerence (SAREF) Ontology*. In Formal Ontologies Meet Industry, pages 100–112, Cham, Switzerland, 2015. Springer. doi:10.1007/978-3-319-21545-7_9.

[20] H. Rijgersberg, M. Van Assem, and J. Top. *Ontology of units of measure and related concepts*. Semantic Web, 4(1):3–13, 2013. doi:10.3233/SW-2012-0069.

[21] G. Rodola. *psutil*, 2022. Accessed: 2023-03-29. Available from: https://github.com/giampaolo/psutil.

[22] *RSP Service Interface for C-SPARQL*. Accessed: 2018-10-18. Available from: https://github.com/streamreasoning/rsp-services-csparql/.

[23] The Apache Software Foundation. *Apache Jena*, 2021. Accessed: 2022-02-01. Available from: https://jena.apache.org/.

[24] J. E. López de Vergara, A. Guerrero, V. A. Villagrá, and J. Berrocal. *Ontology-based network management: study cases and lessons learned*. Journal of Network and Systems Management, 17(3):234–254, 2009. doi:10.1007/s10922-009-9129-1.

[25] P. S. Moraes, L. N. Sampaio, J. A. S. Monteiro, and M. Portnoi. *MonONTO: A Domain Ontology for Network Monitoring and Recommendation for Advanced Internet Applications Users*. In NOMS Workshops 2008 – IEEE Network Operations and Management Symposium Workshops, pages 116–123, New York, NY, USA, 2008. IEEE. doi:10.1109/NOMSW.2007.21.

[26] A. Salvador, J. López de Vergara Méndez, G. Tropea, G. Blefari-Melazzi, and A. Ferreiro. *Ontology design and implementation for IP networks monitoring*. In International Workshop on Web & Semantic Technology (WeST-2009), 2009.

[27] W. Adianto, C. de Laat, and P. Grosso. *Future Internet Ontologies: The NOVI Experience*. Preprint submitted to Semantic Web Journal, 2009. Available from: https://www.semantic-web-journal.net/system/files/swj580.pdf.

[28] R. F. Silva, P. Carvalho, S. Rito Lima, L. Álvarez Sabucedo, J. M. Santos Gago, and J. M. C. Silva. *An Ontology-Based Recommendation System for Context-Aware Network Monitoring*. In Á. Rocha, H. Adeli, L. P. Reis, and S. Costanzo, editors, New Knowledge in Information Systems and Technologies, pages 373–384, Cham, Switzerland, 2019. Springer. doi:10.1007/978-3-030-16184-2_36.

[29] K. Krinkin, A. Vodyaho, I. Kulikov, and N. Zhukova. *Models of Telecommunications Network Monitoring Based on Knowledge Graphs*. In 9th Mediterranean Conference on Embedded Computing (MECO 2020), pages 1–7, 2020. doi:10.1109/MECO49872.2020.9134148.

[30] I. Kulikov, A. Vodyaho, E. Stankova, and N. Zhukova. *Ontology for Knowledge Graphs of Telecommunication Network Monitoring Systems*. In Computational Science and Its Applications – ICCSA 2021, pages 432–446, Cham, Switzerland, 2021. Springer. doi:10.1007/978-3-030-87010-2_32.

[31] L. Fallon, J. Keeney, and D. O'Sullivan. *Applying Semantics to Optimize End-User Services in Telecommunication Networks*. In R. Meersman, H. Panetto, T. Dillon, M. Missikoff, L. Liu, O. Pastor, A. Cuzzocrea, and T. Sellis, editors, On the Move to Meaningful Internet Systems: OTM 2014 Conferences, pages 718–726, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-662-45563-0_44.

[32] W. Funika, M. Janczykowski, K. Jopek, and M. Grzegorczyk. *An Ontology-based Approach to Performance Monitoring of MUSCLE-bound Multi-scale Applications*. Procedia Computer Science, 18:1126–1135, 2013. 2013 International Conference on Computational Science. doi:10.1016/j.procs.2013.05.278.

[33] K. Ryabinin and S. Chuprina. *Ontology-Driven Edge Computing*. In Computational Science – ICCS 2020, pages 312–325, Cham, Switzerland, 2020. Springer. doi:10.1007/978-3-030-50436-6_23.

[34] V. M. Tayur and R. Suchithra. *A Comprehensive Ontology for Internet of Things (COIoT)*. In 2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP), pages 1–6, New York, NY, USA, 2019. IEEE. doi:10.1109/ICACCP.2019.8882936.

[35] J. Strassner. *DEN-ng: achieving business-driven network management*. In NOMS 2002. IEEE/IFIP Network Operations and Management Symposium.'Management Solutions for the New Communications World'(Cat. No. 02CH37327), pages 753–766, New York, NY, USA, 2002. IEEE. doi:10.1109/NOMS.2002.1015622.

[36] J. Famaey, S. Latré, J. Strassner, and F. De Turck. *An Ontology-Driven Semantic Bus for Autonomic Communication Elements*. In Modelling Autonomic Communication Environments, pages 37–50, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-16836-9_4.

[37] S. Latré, J. Famaey, J. Strassner, and F. De Turck. *Automated context dissemination for autonomic collaborative networks through semantic subscription filter generation*. Journal of Network and Computer Applications, 36(6):1405–1417, 2013. doi:10.1016/j.jnca.2013.01.011.

[38] A. K. A. Hwaitat, A. Shaheen, K. Adhim, E. N. Arkebat, and A. A. A. Hwiatat. *Computer Hardware Components Ontology*. Modern Applied Science, 12(3):35–40, 2018. doi:10.5539/mas.v12n3p35.

[39] J. Keeney, L. Fallon, W. Tai, and D. O'Sullivan. *Towards composite semantic reasoning for realtime network management data enrichment*. In 11th International Conference on Network and Service Management (CNSM 2015), pages 246–250, 2015. doi:10.1109/CNSM.2015.7367365.

[40] J. Keeney, D. Lewis, and D. O'Sullivan. *Ontological semantics for distributing contextual knowledge in highly distributed autonomic systems*. Journal of Network and Systems Management, 15(1):75–86, 2007. doi:10.1007/s10922-006-9054-5.

[41] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt. *AIOLOS: Middleware for improving mobile application performance through cyber foraging*. Journal of Systems and Software, 85(11):2629–2639, 2012. doi:10.1016/j.jss.2012.06.011.

[42] M. Sebrechts, B. Volckaert, F. De Turck, K. Yangy, and M. AL-Naday. *Fog Native Architecture: Intent-Based Workflows to Take Cloud Native Towards the Edge*. IEEE Communications Magazine, 60(8):44–50, 2022. doi:10.1109/MCOM.003.2101075.

[43] A. K. Idrees and A. K. M. Al-Qurabat. *Energy-efficient data transmission and aggregation protocol in periodic sensor networks based fog computing*. Journal of Network and Systems Management, 29(1), 2021. doi:10.1007/s10922-020-09567-4.

[44] K. Jaiswal and V. Anand. *A Survey on IoT-Based Healthcare System: Potential Applications, Issues, and Challenges*. In A. A. Rizvanov, B. K. Singh, and P. Ganasala, editors, Advances in Biomedical Engineering and Technology, pages 459–471. Springer Singapore, 2021. doi:10.1007/978-981-15-6329-4_38.

[45] J. Bai, C. Di, L. Xiao, K. R. Evenson, A. Z. LaCroix, C. M. Crainiceanu, and D. M. Buchner. *An activity index for raw accelerometry data and its comparison with other activity metrics*. PLoS ONE, 11(8):e0160644, 2016. doi:10.1371/journal.pone.0160644.

[46] B. Steenwinckel, M. De Brouwer, M. Stojchevska, J. Van Der Donckt, J. Nelis, J. Ruyssinck, J. van der Herten, K. Casier, J. Van Ooteghem, P. Crombez, F. De Turck, S. Van Hoecke, and F. Ongenae. *Data Analytics For Health and Connected Care: Ontology, Knowledge Graph and Applications*. In Proceedings of the 16th EAI International Conference on Pervasive Computing Technologies for Healthcare (EAI PervasiveHealth 2022), 2022. Available from: https://dahcc.idlab.ugent.be.

[47] M. Girod-Genet, L. N. Ismail, M. Lefrançois, and J. Moreira. *ETSI TS 103 410-8 V1.1.1 (2020-07): SmartM2M; Extension to SAREF; Part 8: eHealth/Ageing-well Domain*. Technical report, ETSI Technical Committee Smart Machine-to-Machine communications (SmartM2M), 2020. Available from: https://www.etsi.org/deliver/etsi_ts/103400_103499/10341008/01.01.01_60/ts_10341008v010101p.pdf.

[48] I. Esnaola-Gonzalez, J. Bermúdez, I. Fernández, and A. Arnaiz. *Two Ontology Design Patterns toward Energy Efficiency in Buildings*. In Proceedings of the 9th Workshop on Ontology Design and Patterns (WOP 2018), co-located with 17th International Semantic Web Conference (ISWC 2018), pages 14–28. CEUR

Workshop Proceedings, 2018. Available from: https://ceur-ws.org/Vol-2195/pattern_paper_2.pdf.

[49] Empatica. *E4 wristband*, 2020. Accessed: 2020-10-23. Available from: https://www.empatica.com/research/e4.

[50] Ghent University – imec. *iLab.t Virtual Wall*, 2023. Accessed: 2023-04-09. Available from: https://doc.ilabt.imec.be/ilabt/virtualwall.

Addendum 5.A Examples of the DIVIDE Meta Model triples

This addendum gives some examples of how the DIVIDE meta-information and monitoring observations can be represented in the DIVIDE Meta Model with the modules of the Meta Model ontology:

- Listing 5.8 shows how the relevant meta-information of DIVIDE is semantically described using the concepts of the `DivideCore` ontology module.
- Listing 5.9 illustrates how a measurement of the average execution time of an RSP query can be semantically described with the concepts of the `Monitoring` ontology module.

Listing 5.8: Example of how the `DivideCore` module of the Meta Model ontology is used to model all relevant meta-information about DIVIDE in the DIVIDE Meta Model (part 1/2). This example considers a DIVIDE engine with one DIVIDE query and one DIVIDE component. One RSP query derived from this DIVIDE query is registered to the Local RSP Engine of this DIVIDE component. The triples are presented in RDF/Turtle syntax.

```
# additional, temporary prefixes to make this listing more readable
@prefix divide-engine: <https://divide.idlab.ugent.be/meta-model/entity/engine/> .
@prefix divide-component: <https://divide.idlab.ugent.be/meta-model/entity/component/> .
@prefix divide-query: <https://divide.idlab.ugent.be/meta-model/entity/divide-query/> .
@prefix rsp-engine: <https://divide.idlab.ugent.be/meta-model/entity/rsp-engine/> .
@prefix device: <https://divide.idlab.ugent.be/meta-model/entity/device/> .

# DIVIDE engine
divide-engine:44c52eb3-3a03-4d11-8c96-e1854196e4e7
    a divide-core:DivideEngine ;
    divide-core:hasID "44c52eb3-3a03-4d11-8c96-e1854196e4e7" ;
    divide-core:isHostedBy device:10.42.0.112 .

# DIVIDE component
divide-component:10.42.0.35-8100-
    a divide-core:DivideComponent ;
    divide-core:hasID "10.42.0.35-8100-" ;
    divide-core:hasCentralRspEngine rsp-engine:central ;
    divide-core:hasLocalRspEngine rsp-engine:10.42.0.35-8100- ;
    divide-core:isHostedBy device:10.42.0.35 .

# device of DIVIDE component
device:10.42.0.35 a saref-core:Device ;
    divide-core:hasIPAddress "10.42.0.35" .

# Local RSP Engine of DIVIDE component
rsp-engine:10.42.0.35-8100- a divide-core:RspEngine ;
    divide-core:hasServerPort "8100"^^xsd:integer .

# DIVIDE query
divide-query:activity-index a divide-core:DivideQuery ;
    divide-core:hasName "activity-index" ;
    divide-core:hasQueryDeployment <https://divide.idlab.ugent.be/meta-model/entity/divide-
          query/activity-index/deployment/10.42.0.35-8100-> .
```

Listing 5.8: Example of how the `DivideCore` module of the Meta Model ontology is used to model all relevant meta-information about DIVIDE in the DIVIDE Meta Model (part 2/2). This example considers a DIVIDE engine with one DIVIDE query and one DIVIDE component. One RSP query derived from this DIVIDE query is registered to the Local RSP Engine of this DIVIDE component. The triples are presented in RDF/Turtle syntax.

```
# RSP query on DIVIDE component, derived from DIVIDE query
<https://divide.idlab.ugent.be/meta-model/entity/rsp-engine/10.42.0.35-8100-/rsp-query/10f67219
      -36c9-4bda-97f4-dc9420537348>
    a divide-core:RspQuery ;
    divide-core:hasName "Q0mmlll" ;
    divide-core:hasID "10f67219-36c9-4bda-97f4-dc9420537348" ;
    divide-core:hasAssociatedComponent divide-component:10.42.0.35-8100- ;
    divide-core:hasCorrespondingDivideQuery divide-query:activity-index ;
    divide-core:isRegisteredTo rsp-engine:10.42.0.35-8100- ;
    divide-core:hasWindowSizeInSeconds "80"^^xsd:integer ;
    divide-core:hasQuerySlidingStepInSeconds "10"^^xsd:integer ;
    divide-core:hasInputStreamWindow <https://divide.idlab.ugent.be/meta-model/entity/rsp-
          engine/10.42.0.35-8100-/rsp-query/10f67219-36c9-4bda-97f4-dc9420537348/stream-window/
          http%3A%2F%2Fprotego.ilabt.imec.be%2Fidlab.homelab-RANGE+80s+STEP+10s> .

# local deployment of RSP query
<https://divide.idlab.ugent.be/meta-model/entity/divide-query/activity-index/deployment
      /10.42.0.35-8100->
    a divide-core:QueryDeployment ;
    saref-core:isAbout divide-component:10.42.0.35-8100- ;
    divide-core:hasQueryLocation <https://divide.idlab.ugent.be/meta-model/entity/divide-query/
          activity-index/deployment/10.42.0.35-8100-/location/LocalLocation> .
<https://divide.idlab.ugent.be/meta-model/entity/divide-query/activity-index/deployment
      /10.42.0.35-8100-/location/LocalLocation>
    a divide-core:LocalLocation .

# stream window of RSP query
<https://divide.idlab.ugent.be/meta-model/entity/rsp-engine/10.42.0.35-8100-/rsp-query/10f67219
      -36c9-4bda-97f4-dc9420537348/stream-window/http%3A%2F%2Fprotego.ilabt.imec.be%2Fidlab.
      homelab-RANGE+80s+STEP+10s>
    a divide-core:StreamWindow ;
    divide-core:hasInputStream <https://divide.idlab.ugent.be/meta-model/entity/rsp-engine
          /10.42.0.35-8100-/rdf-stream/http%3A%2F%2Fprotego.ilabt.imec.be%2Fidlab.homelab> ;
    divide-core:hasWindowDefinition "RANGE 80s STEP 10s" ;
    divide-core:hasWindowSizeInSeconds "80"^^xsd:integer ;
    divide-core:hasQuerySlidingStepInSeconds "10"^^xsd:integer .

# RDF stream in stream window of RSP query
<https://divide.idlab.ugent.be/meta-model/entity/rsp-engine/10.42.0.35-8100-/rdf-stream/http%3A
      %2F%2Fprotego.ilabt.imec.be%2Fidlab.homelab>
    a divide-core:RdfStream ;
    divide-core:hasStreamName
        "http://protego.ilabt.imec.be/idlab.homelab" .
```

Listing 5.9: Example of how the `Monitoring` module of the DIVIDE Meta Model ontology allows modeling the average execution time of an RSP query, represented in RDF/Turtle syntax

```
<https://divide.idlab.ugent.be/10.42.0.35-8100-/rsp_query_execution_time/avg/obs64>
    a saref-core:Measurement ;
    saref-core:hasValue "0.561"^^xsd:float ;
    om:hasAggregateFunction om:average ;
    saref-core:hasTimestamp "2023-04-02T08:33:22"^^xsd:dateTime ;
    saref-core:isMeasuredIn om:second-Time ;
    saref-core:relatesToProperty [ a monitoring:RspQueryExecutionTime ] ;
    saref-core:isMeasurementOf  <https://divide.idlab.ugent.be/meta-model/entity/rsp-engine
          /10.42.0.35-8100-/rsp-query/a76f39a8-ff76-453d-9589-a10b7d6ea942> .
```

Addendum 5.B Additional implementation details

This addendum provides some additional details about our implementation of the DIVIDE Local Monitor:

- Listing 5.10 shows an example configuration of the DIVIDE Local Monitor.
- Listing 5.11 presents the C-SPARQL aggregation query that is deployed on our implementation of the Local Monitor RSP Engine.

Listing 5.10: Example JSON configuration of the DIVIDE Local Monitor

```
{
  "component_id": "10.10.145.9-8175-",
  "device_id": "10.10.145.9",
  "monitor": {
    "rsp": true,
    "network": true,
    "device": true
  },
  "local": {
    "rsp_engine": {
      "monitor": {
        "ws_port": 54548
      }
    },
    "public_network_interface": "wlp1s0"
  },
  "central": {
    "monitor_reasoning_service": {
      "protocol": "http",
      "host": "10.10.145.233",
      "port": 54555,
      "uri": "/globalmonitorreasoningservice"
    }
  }
}
```

Listing 5.11: C-SPARQL aggregation query that is continuously evaluated on the implementation of the DIVIDE Local Monitor RSP engine

```
CONSTRUCT {
    [ a saref-core:Measurement ;
        saref-core:hasValue ?minV ;
        om:hasAggregateFunction om:minimum ;
        saref-core:isMeasuredIn ?unit ;
        saref-core:relatesToProperty [ a ?prop ] ;
        saref-core:isMeasurementOf ?featureOfInterest ;
        saref-core:hasTimestamp ?now ] .

    [ a saref-core:Measurement ;
        saref-core:hasValue ?maxV ;
        om:hasAggregateFunction om:maximum ;
        saref-core:isMeasuredIn ?unit ;
        saref-core:relatesToProperty [ a ?prop ] ;
        saref-core:isMeasurementOf ?featureOfInterest ;
        saref-core:hasTimestamp ?now ] .

    [ a saref-core:Measurement ;
        saref-core:hasValue ?avgV ;
        om:hasAggregateFunction om:average ;
        saref-core:isMeasuredIn ?unit ;
        saref-core:relatesToProperty [ a ?prop ] ;
        saref-core:isMeasurementOf ?featureOfInterest ;
        saref-core:hasTimestamp ?now ] .
}
FROM STREAM <https://divide.idlab.ugent.be/monitor/local> [RANGE 60s STEP 20s]
WHERE {
    BIND (NOW() as ?now)
    {
        SELECT ?featureOfInterest ?prop ?unit
               (MIN(?v) AS ?minV)
               (MAX(?v) AS ?maxV)
               (AVG(?v) AS ?avgV)
        WHERE {
            ?m a saref-core:Measurement ;
               saref-core:hasValue ?v ;
               saref-core:isMeasuredIn ?unit ;
               saref-core:relatesToProperty [ a ?prop ] ;
               saref-core:isMeasurementOf ?featureOfInterest .
        }
        GROUP BY ?featureOfInterest ?prop ?unit
    }
}
```

Addendum 5.C Semantic details of the evaluation use case

This addendum provides the semantic details about the homecare monitoring use case of the evaluations performed in this chapter, which is discussed in Section 5.6.1:

- Listing 5.12 gives an overview of additional prefixes used in the listings of this addendum. This concerns prefixes specific to the evaluation use case that were not yet listed in Listing 5.1.
- Listing 5.13 and 5.14 present relevant definitions contained in the additional ontology module of the DAHCC ontology that is designed for the evaluations.
- Listing 5.15 presents the most important triples in the use case context of the described evaluation scenarios.
- Listing 5.17 and 5.16 present the DIVIDE query that is added to DIVIDE Core during the executed evaluation scenarios. This DIVIDE query is used by DIVIDE to derive the RSP query that continuously measures the activity index of the monitored patient. Listing 5.17 presents the sensor query rule of this DIVIDE query, while Listing 5.16 presents its goal.

Listing 5.12: Semantic description of the additional prefixes used in the listings of this addendum

```
# DAHCC ontology including the additional ontology module
@prefix ActivityRecognition: <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/> .
@prefix MonitoredPerson: <https://dahcc.idlab.ugent.be/Ontology/MonitoredPerson/> .
@prefix Sensors: <https://dahcc.idlab.ugent.be/Ontology/Sensors/> .
@prefix SensorsAndWearables: <https://dahcc.idlab.ugent.be/Ontology/SensorsAndWearables/> .
@prefix HealthParameterCalculation:
    <https://dahcc.idlab.ugent.be/Ontology/HealthParameterCalculation/> .

# instances in use case scenario
@prefix : <http://divide.ilabt.imec.be/idlab.homelab/> .
@prefix patients: <http://divide.ilabt.imec.be/idlab.homelab/patients/> .
@prefix Homelab: <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/> .
@prefix HomelabWearable: <https://dahcc.idlab.ugent.be/Homelab/SensorsAndWearables/> .

# additional imports of DAHCC ontology modules
@prefix saref4ehaw: <https://saref.etsi.org/saref4ehaw/> .
@prefix time: <http://www.w3.org/2006/time#> .

# definitions within DIVIDE
@prefix sd: <http://idlab.ugent.be/sensdesc#> .
@prefix sd-query: <http://idlab.ugent.be/sensdesc/query#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
```

Listing 5.13: Overview of how different subclass and equivalence relations are defined in the additional ontology module of the DAHCC ontology created for the evaluations in this chapter. These definitions allow a semantic reasoner to derive when a certain health parameter is relevant to be monitored for a patient. To improve readability, all definitions are listed in Manchester syntax and the `HealthParameterCalculation:` prefix is replaced by the `:` prefix.

```
:ActivityIndex SubClassOf: :HealthParameter

:RelevantActivityIndex SubClassOf: :ActivityIndex
:RelevantActivityIndex EquivalentTo:
    :ActivityIndex and
    (:calculationMadeFor some :PatientThatRequiresMovementMonitoring)

:PatientThatRequiresMovementMonitoring EquivalentTo:
    saref4ehaw:Patient and
    (saref4ehaw:hasChronicDisease some (
        saref4ehaw:ChronicDisease and
        (:requiresMovementMonitoring value true)))
```

Listing 5.14: Overview of important ontology definitions in the additional ontology module of the DAHCC ontology created for the evaluations in this chapter, presented in RDF/Turtle syntax. These definitions describe a health parameter calculator system that allows calculating a patient's activity index whenever relevant. To improve readability, the `HealthParameterCalculation:` prefix is replaced by the `:` prefix.

```
# define health parameter calculator
:HealthParameterCalculator rdf:type :Calculator ;
    eep:implements :HealthParameterCalculatorConfig1 .
:HealthParameterCalculatorConfig1
    rdf:type ActivityRecognition:Configuration ;
    :containsHealthParameterConfig :activity_index_config .

# define that calculation of activity index should be done
# using the wearable acceleration property
:activity_index_config rdf:type :HealthParameterConfig ;
    :forHealthParameter [ rdf:type :ActivityIndex ] ;
    :forProperty [ rdf:type SensorsAndWearables:WearableAcceleration ] .

# define different medical conditions that require movement monitoring
:HeartDisease rdf:type saref4ehaw:ChronicDisease ;
    :requiresMovementMonitoring "true"^^xsd:boolean .
:HighFallRisk rdf:type saref4ehaw:ChronicDisease ;
    :requiresMovementMonitoring "true"^^xsd:boolean .
:Dementia rdf:type saref4ehaw:ChronicDisease ;
    :requiresMovementMonitoring "true"^^xsd:boolean .
```

Listing 5.15: Overview of the most important triples in the use case context of the described evaluation scenarios

```
# patient lives in HomeLab
patients:patient373 rdf:type saref4ehaw:Patient ;
    MonitoredPerson:livesIn Homelab:homelab .

# patient has an Empatica wearable
patients:patient373 rdf:type saref4wear:Wearer .
HomelabWearable:empatica.E4.A03813
    saref4wear:isLocatedOn patients:patient373 ;
    MonitoredPerson:hasLocation Homelab:homelab .

# patient has a heart disease
patients:patient373 saref4ehaw:hasChronicDisease
    HealthParameterCalculation:HeartDisease .
```

Listing 5.16: Goal of the internal representation of the DIVIDE query that is used in the evaluation scenarios to derive an RSP query that continuously measures a patient's activity index. This goal contains the semantic output that should be filtered by the RSP queries derived from this DIVIDE query.

```
{
    ?p rdf:type HealthParameterCalculation:RelevantActivityIndex ;
       saref-core:hasValue ?v ;
       HealthParameterCalculation:calculationMadeFor ?patient ;
       HealthParameterCalculation:calculatedBy ?calculator ;
       saref-core:hasTimestamp ?t .
} => {
    _:p rdf:type HealthParameterCalculation:RelevantActivityIndex ;
        saref-core:hasValue ?v ;
        HealthParameterCalculation:calculationMadeFor ?patient ;
        HealthParameterCalculation:calculatedBy ?calculator ;
        saref-core:hasTimestamp ?t .
} .
```

Listing 5.17: Sensor query rule of the internal representation of the DIVIDE query that is used in the evaluation scenarios to derive an RSP query that continuously measures a patient's activity index (part 1/2). The sensor query rule also includes the template of this RSP query in RSP-QL syntax. During the query derivation, DIVIDE substitutes the input variables and window parameters into this template.

```
{
    ?calculator rdf:type HealthParameterCalculation:Calculator ;
        <https://w3id.org/eep#implements> [
            rdf:type ActivityRecognition:Configuration ;
            HealthParameterCalculation:containsHealthParameterConfig ?hpc ] .
    ?hpc rdf:type HealthParameterCalculation:HealthParameterConfig ;
        HealthParameterCalculation:forHealthParameter [
            rdf:type HealthParameterCalculation:ActivityIndex ] ;
        HealthParameterCalculation:forProperty [ rdf:type ?prop ] .

    ?prop rdfs:subClassOf HealthParameterCalculation:ConditionableProperty .

    ?sensor rdf:type saref-core:Device ;
        saref-core:measuresProperty [
            rdf:type ?prop ;
            SensorsAndWearables:hasAxis [
                rdf:type SensorsAndWearables:XAxis ] ] ;
        Sensors:analyseStateOf ?patient ;
        MonitoredPerson:hasLocation ?home .

    ?patient rdf:type saref4ehaw:Patient ; MonitoredPerson:livesIn ?home .
}
=>
{
    _:q rdf:type sd:Query ; sd:pattern sd-query:pattern ;
        sd:inputVariables (("?s_iri" ?sensor) ("?patient" ?patient)
                           ("?calculator" ?calculator)) ;
        sd:windowParameters (("?range" 80 time:seconds)
                             ("?slide" 10 time:seconds)) .

    _:p rdf:type HealthParameterCalculation:ActivityIndex ;
        HealthParameterCalculation:calculationMadeFor ?patient ;
        HealthParameterCalculation:calculatedBy ?calculator ;
        saref-core:hasTimestamp _:t ; saref-core:hasValue _:v .
} .
```

Listing 5.17: Sensor query rule of the internal representation of the DIVIDE query that is used in the evaluation scenarios to derive an RSP query that continuously measures a patient's activity index (part 2/2). The sensor query rule also includes the template of this RSP query in RSP-QL syntax. During the query derivation, DIVIDE substitutes the input variables and window parameters into this template.

```
sd-query:prefixes-activity-index rdf:type owl:Ontology ;
    sh:declare [ sh:prefix "xsd" ;
                 sh:namespace "http://www.w3.org/2001/XMLSchema#"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "saref-core" ;
                 sh:namespace "https://saref.etsi.org/core/"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "ActivityRecognition" ; sh:namespace "https://dahcc.idlab.ugent.be/
          Ontology/ActivityRecognition/"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "HealthParameterCalculation" ; sh:namespace "https://dahcc.idlab.
          ugent.be/Ontology/HealthParameterCalculation/"^^xsd:anyURI ] .

sd-query:pattern-activity-index
    rdf:type sd:QueryPattern ;
    sh:prefixes sd-query:prefixes-activity-index ;
    sh:construct """
        CONSTRUCT {
            _:p a HealthParameterCalculation:RelevantActivityIndex ;
                saref-core:hasValue ?ai ;
                HealthParameterCalculation:calculationMadeFor ?patient ;
                HealthParameterCalculation:calculatedBy ?calculator ;
                saref-core:hasTimestamp ?now .
        }
        FROM NAMED WINDOW :win ON <http://protego.ilabt.imec.be/idlab.homelab>
            [RANGE ?{range} SLIDE ?{slide}]
        WHERE { WINDOW :win {
            BIND (NOW() AS ?now)

            { SELECT ?sensor (AVG(?var) AS ?ai)
              WHERE {
                  SELECT ?sensor ?p (IF(?count=1, -1, (?sx / ?count)) AS ?var)
                  WHERE {
                      SELECT ?sensor ?p (SUM(?x) AS ?sx) ?count
                      WHERE {
                          SELECT ?sensor ?p ?v (xsd:float(?v) AS ?ni) ?mean
                                 (xsd:float(?ni - ?mean) AS ?nmean)
                                 (((?nmean) * (?nmean)) AS ?x) ?count
                          WHERE {
                              ?sensor saref-core:makesMeasurement [
                                  saref-core:hasValue ?v ;
                                  saref-core:relatesToProperty ?p ] .
                              { SELECT ?p (?s_iri AS ?sensor)
                                       (AVG(?v2) AS ?mean)
                                       (COUNT(?v2) as ?count)
                                WHERE {
                                    ?s_iri saref-core:makesMeasurement [
                                        saref-core:hasValue ?v2 ;
                                        saref-core:relatesToProperty ?p ] .
                                }
                                GROUP BY (?s_iri AS ?sensor) ?p }
                          }
                      } GROUP BY ?sensor ?p ?count
                  }
              }
              GROUP BY ?sensor }
        } }
        LIMIT 1
        """ .
```

Addendum 5.D Additional results of the evaluation of DIVIDE

This addendum provides additional insights in the evaluation results of this chapter, which are presented in Section 5.7.

- The results in this chapter are aggregated over multiple evaluation runs. However, when running the same scenario multiple times on the DIVIDE Monitoring set-up, different patterns can sometimes be observed. This is the case for both the results in Section 5.7.1 (Figure 5.4a) and Section 5.7.2 (Figures 5.6a, 5.7a and 5.7b). We define a *pattern* as the sequence of RSP query executions combined with the query window parameters for the results in Section 5.7.1, and the sequence of RSP query executions combined with their query location in Section 5.7.1. Since the timeline visualizations for the DIVIDE Monitoring set-up are aggregated in time as well, it is impossible to aggregate different result patterns in one figure. Hence, the results in those figures aggregate the runs with the result pattern that occurred most frequently.

  To illustrate this with an example, consider the results for a single evaluation run of the second evaluation scenario on the DIVIDE Monitoring set-up in Figure 5.11. This run is not aggregated in the results of Figure 5.6a, since it has a different result pattern: after the first 13 query executions on the Central RSP Engine, it has only *10* query executions on the Local RSP Engine, before the query is moved again to the Central RSP Engine. In contrast, the aggregated evaluation runs in Figure 5.6a start with 13 query executions on the Central RSP Engine, followed by *12* executions on the Local RSP Engine.

- Figure 5.12 shows the results for a single evaluation run of the second evaluation scenario on the DIVIDE Central set-up. This scenario updates the RSP query location based on the monitored network RTT. This figure represents one of the runs that are included in the aggregated results of Figure 5.6c. It demonstrates that the simulated network capacity restrictions can lead to accumulating delays in the processing of the data streams, if the RSP query remains registered to the Central RSP Engine. For multiple executions of the RSP query, the network overhead is 25 seconds or higher, up to almost 33 seconds at its highest point.

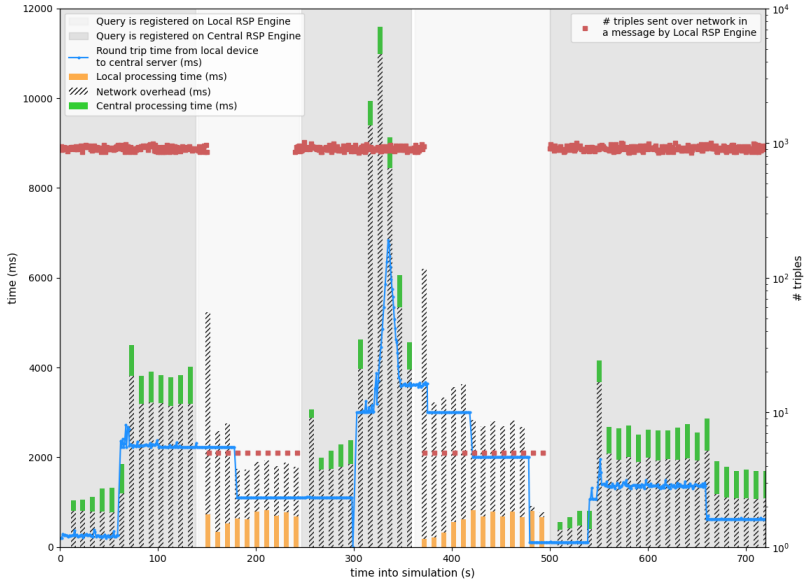


Figure 5.11: Results of a single evaluation run of evaluation scenario 2 that updates the RSP query location based on the monitored network RTT, for the DIVIDE Monitoring set-up. This run is not included in the aggregated results of Figure 5.6a, because it has a different result pattern.

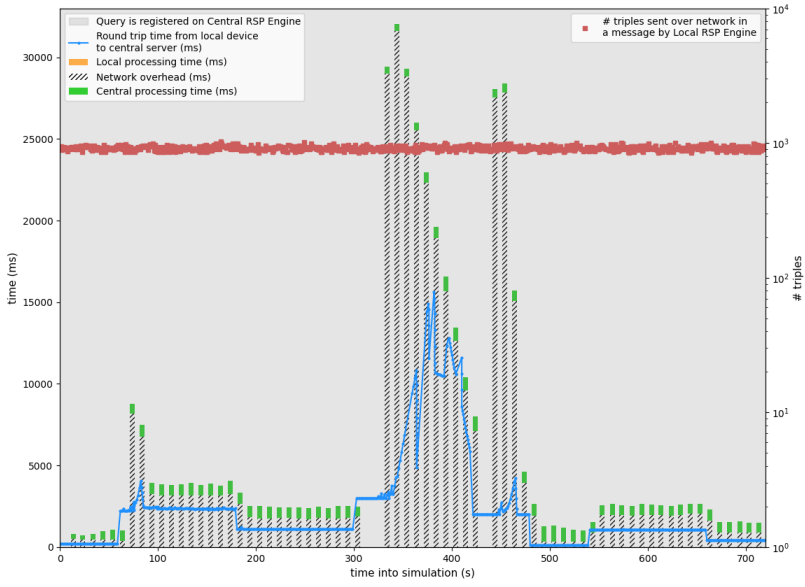


Figure 5.12: Results of a single evaluation run of evaluation scenario 2 that updates the RSP query location based on the monitored network RTT, for the DIVIDE Central set-up. This run is included in the aggregated results shown in Figure 5.6c.

6

Optimized Continuous Homecare Provisioning through Distributed Data-Driven Semantic Services and Cross-Organizational Workflows

In the previous chapters, a generic cascading reasoning framework and an additional semantic IoT platform component called DIVIDE were designed. The evaluation use cases in these chapters mainly focused on the queries that are continuously evaluated on the stream reasoning engines in the platform. In this chapter, DIVIDE is embedded into a full semantic platform. It presents a reference architecture that can be mapped to the generic design of the cascading reasoning framework presented in Chapter 2. By coupling data-driven semantic services to the outputs derived by the stream processing queries, and by integrating an engine that composes semantic workflows, the chapter shows specifically for the healthcare domain how continuous (home)care can be optimized on different levels and across stakeholders involved in the patient's care. To this end, the chapter also presents a use case demonstrator about the homecare monitoring use case UC2, which mainly focuses in this chapter on smart monitoring based on a patient's medical profile, and the construction and cross-organizational coordination of treatment plans to a patient's diagnoses.

This chapter addresses research challenge RCH4 ("Closing the loop by embedding the solutions into a full semantic platform that is efficient & performant") by discussing research contribution RC04. It validates research hypothesis RH6: "A semantic IoT platform component that adaptively manages and configures queries according to varying environmental context, can be embedded in a semantic platform with other semantic components that define and construct data-driven

semantic services and cross-organizational semantic workflows. Put together, the resulting cascading reasoning architecture can be leveraged to optimize relevant IoT use cases.".

⋆ ⋆ ⋆

M. De Brouwer, P. Bonte, D. Arndt, M. Vander Sande, A. Dimou, R. Verborgh, F. De Turck, and F. Ongenae

Submitted for review to Journal of Biomedical Semantics, August 2023.

Abstract

Background. In healthcare, an increasing collaboration can be noticed between different caregivers, especially considering the shift to homecare. To provide optimal patient care, efficient coordination of data and workflows between these different stakeholders is required. To achieve this, data should be exposed in a machine-interpretable, reusable manner. In addition, there is a need for smart, dynamic, personalized and performant services provided on top of this data. Flexible workflows should be defined that realize their desired functionality, adhere to use case specific quality constraints and improve coordination across stakeholders. User interfaces should allow configuring all of this in an easy, user-friendly way.

Methods. Existing tools built upon Semantic Web technologies can resolve the imposed challenges by providing data-driven semantic services and constructing cross-organizational workflows. These tools include RMLStreamer to generate Linked Data, DIVIDE to adaptively manage contextually relevant local queries, Streaming MASSIF to deploy reusable services, AMADEUS to compose semantic workflows, and RMLEditor and Matey to configure rules to generate Linked Data. They are brought together in a distributed, cascading reasoning architecture. A use case demonstrator is built on a scenario that focuses on personalized smart monitoring and cross-organizational treatment planning.

Results. The performance of the demonstrator's implementation is evaluated. The averaged results show that the monitoring pipeline efficiently processes a data stream of 14 incoming observations per second: RMLStreamer maps JSON observations to RDF in 13.5 ms, a C-SPARQL query to generate fever alarms is executed on a window of 5 s in 26.4 ms, and Streaming MASSIF generates a smart notification for fever alarms based on severity and urgency in 1539.5 ms. DIVIDE derives the three demonstrator C-SPARQL queries in 7249.5 ms, while AMADEUS constructs a colon cancer treatment plan and performs conflict detection with the resulting plan in 190.8 ms and 1335.7 ms, respectively.

Conclusions. Existing tools built upon Semantic Web technologies can be leveraged to optimize continuous care provisioning. The evaluation of the building blocks on a realistic homecare monitoring use case demonstrates their good performance and

applicability. Further extending the available user interfaces for some tools such as DIVIDE is required to increase their adoption.

6.1 Background

6.1.1 Introduction

Due to increased digitization allowing more easily capturing relevant data, industries are faced with the challenge of processing an increase in complex and heterogeneous data in an automated, scalable, performant and cost-efficient manner. Increasing demand can also be noted for offering more personalized, context-aware and intelligent applications to end users [1, 2]. This translates into increased non-recurring engineering costs, a need to build custom interfaces and a long time to market.

To tackle these issues, companies typically adopt a Service Oriented Architecture (SOA) in which systems are made up out of a set of services each offering a self-contained unit of functionality [3]. By combining services into workflows, the required functionality can be offered in a structured way [4]. Services can easily be reused in different workflows, resulting in lower development and maintenance costs and a quicker time to market. During the last years, an explosion of such services can be observed.

Existing workflow engines try to make the reuse of provided services manageable. Nevertheless, usually, workflows are created in a manual way. In addition, the intelligence of a platform is mostly distributed over the workflow engine and software code. As a consequence, management becomes an immense burden. These workflows and individual services are also faced with increasing scalability and performance issues incurred by the massive amounts of data they need to process, such as the data generated by various sensors in an Internet of Things (IoT) environment. Finally, existing end user tools that enable domain experts to create services and workflows do not scale in a non-entertainment or educational setting.

In healthcare, the aforementioned general challenges are present as well [5]. A relevant is example is homecare, which has become increasingly important over the last years. This is due to the gradual shift from acute to chronic care, where people are living longer with one or more chronic diseases, requiring more complex care [6]. To reserve residential care for patients with more severe care needs, there is an increasing trend towards shortening hospital stays, by making care delivery more transmural and enabling recovery at home and in service flats. To facilitate this shift to homecare, it is crucial to monitor and follow up the elderly at home in a dependable, accurate manner.

Multiple formal and informal caregivers are involved in following up the condition of homecare patients and making sure they can stay safely at home as long as possible. This is visually illustrated in Figure 6.1. Typically, the patients and the service flats in which they live are equipped with multiple sensors to monitor the patient and environment, and devices to steer the home conditions such as its temperature and lights.

Figure 6.1: Visual overview of different possible caregivers that can be involved in following up homecare patients. These possible formal and informal caregivers can include a nurse, a general practitioner, hospital doctors, family members, etc. The data needed by these caregivers to optimally perform their care tasks and follow up the patient's condition, is typically spread across the various involved stakeholders.

The former may include environmental sensors, localization devices, wearables, medical sensors, and more. Alarms are commonly generated by the sensors, devices and services when anomalous situations are observed, such as an abnormal blood pressure or heart rate. Moreover, patients have a Personal Alarm System (PAS), which they can use to make calls. A nurse regularly visits patients to handle the alarms and calls, and perform daily care tasks and medical follow-up. In addition, patients are followed up by their General Practitioner (GP), are registered as known patients in their hospital, and often also have some informal caregivers such as a family member or friend who regularly check up on them. To perform their care tasks, the caregivers of a given patient depend on the existing patient data and the data gathered by these different sources. However, this data is spread across the various involved stakeholders: the sensor providers have the raw gathered data, the nursing organization sees the incoming calls & alarms and knows some background information about the patient, the GP knows the patient's medical history, while the hospital has even more detailed medical knowledge about the patient. Therefore, it is an existing challenge for these different stakeholders to organize themselves across organizations and leverage all available data optimally, in order to provide the best possible care for their patients.

Looking at these challenges from a technical perspective, four different roles can be discerned: data providers, service providers, integrators and installers. Data providers expose the available data, on which service providers can build services used by the integrators to compose workflows. The installers are the people responsible for configuring all the services, workflows and data provisioning tools to the needs of the patients and caregivers. In typical existing systems, multiple issues exist with respect to these different roles. Custom APIs are typically built to expose each data source, on which custom services are built that are configured into generic, static workflows that fulfill a particular need. This means that every time a

new data source or service becomes available, much manual effort is needed by the installers to integrate it. Hence, high costs and development time are required for each cross-organizational workflow that needs to be set up. This also leads to a long time to market. Moreover, the manual convoluted configuration leads to errors and custom APIs and services that cannot be reused. In addition, the custom services lack a personalized approach: available knowledge about patient profiles cannot be efficiently exchanged and exploited, causing generic decisions to be made. An example in homecare monitoring is the assignment of the appropriate caregiver to handle an alarm or call, which is commonly solved in a generic, non-personalized way. Finally, the custom monitoring services usually follow a naive, static, centralized approach: they process all exposed sensor data on central servers, which is especially challenging for high-volume and high-velocity data sources. This reduces performance, scalability, local autonomy and data privacy of the set-up.

To summarize, for each of the given roles, the presented technical issues impose the following challenges for a possible solution:

(a) **Data providers:** How can they easily expose their data to other organizations in a reusable fashion, while making the explicit meaning of the data clear?

(b) **Service providers:** How can they step away from custom & non-reusable services, towards flexible & reusable services that can easily be constructed and configured based on the incoming data? How can they take all available background knowledge and contextual data of the patient profiles into account to become personalized services? How can they intelligently deal with the huge amount of high-volume and high-velocity data coming in and still offer services that meet the use case specific quality requirements such as performance, scalability, local autonomy and data privacy? How can they make the meaning of their services clear such that they can be easily picked up and reused by other organizations?

(c) **Integrators:** How can they move away from static, manually constructed workflows towards flexible workflows, that can easily be configured and adapted based on the available services? How can they dynamically realize the desired functionality and adhere to use case specific quality constraints?

(d) **Installers:** How can they easily expose data and build services in a user-friendly way that minimizes the required configuration effort and the probability of configuration errors?

6.1.2 Semantic Web technologies

To resolve the individual challenges imposed to the different technical roles presented in Section 6.1.1, Semantic Web technologies can be leveraged.

In general, semantics allow a system to semantically annotate different heterogeneous data sources in a common, uniform, machine-interpretable format. In the context of IoT environments, this allows the integration of sensor data with various sources of domain knowledge, background knowledge and context information. By integrating all data, their meaning and context becomes clear, allowing personalized services to process the data and reason on it [7].

Semantic Web technologies represent the vision of the World Wide Web Consortium (W3C) about applying semantics to the web. They consist of a collection of recommended technologies that are perfectly suited to address the challenges in continuous homecare. From these challenges, it follows that there is a need to exchange the available healthcare data across different parties such as a nursing organization, hospital, GP, etc. These different parties need to integrate this healthcare data from various sources and in various formats, and intelligently process it to offer these context-aware, personalized services that meet different quality requirements and to construct dynamic workflows. To do this, machines need to understand the healthcare data in an unambiguous manner, in a similar way as humans do. Moreover, they need techniques to integrate this healthcare data, query it, reason over it, process the data streams generated by the sensors in the patients' environments, and more. Semantic Web technologies apply semantics to provide support for these various tasks to machines, which makes them the ideal solution.

In addition, Semantic Web technologies also allow building declarative solutions. This is an important requirement for solutions that tackle the challenges addressed in this chapter: whenever you want to instrument certain processes or actions based on the healthcare data, you mainly want to declare in a general way *what* should happen, without already hard coding *how* this will happen. Semantic Web technologies make this possible, as they allow expressing the desired actions (e.g., instructions to generate semantic data, queries, services, workflow steps) in a declarative way, independently of their exact implementation.

Semantic Web technologies include the Resource Description Framework (RDF) [8] and the Web Ontology Language (OWL) [9]. They enable semantic enrichment through ontologies, which are semantic models that describe domain concepts, relationships and attributes in a formal way [7, 10]. Different formats exist to store RDF data, such as RDF/Turtle and N-Triples. The collection of RDF datasets on the web that are semantically interconnected is referred to as Linked Data [11]. The SPARQL Protocol and RDF Query Language (SPARQL) [12] is another Semantic Web technology that is used to query RDF data sources. Semantic reasoners such as RDFox [13] and VLog [14] can derive new knowledge based on semantic descriptions and axioms defined in ontologies. Stream reasoning is the research field that investigates the adoption of such semantic reasoning techniques for streaming data [15]. RDF Stream Processing (RSP) engines process RDF data streams by continuously evaluating queries on a data window placed on top of

semantic data streams [16]. RSP-QL is a reference model unifying the semantics of different existing RSP approaches [17].

6.1.3 Objective and chapter organization

The objective of this chapter is to demonstrate how Semantic Web technologies can be leveraged to solve the individual challenges imposed to the different technical roles, as presented in Section 6.1.1. The chapter presents an overall stack of existing tools built on Semantic Web technologies that optimizes continuous homecare provisioning through distributed, data-driven semantic services and dynamic, easily configurable, cross-organizational semantic workflows. For every identified role, the presented tools should solve the imposed challenges in a performant manner by building further on existing Semantic Web technologies. This translates the challenges into the following hypotheses:

(a) **Data providers:** By exposing their data as Linked Data, the meaning and context of the data becomes clear. This way, the data can easily be reused by different services.

(b) **Service providers:** (i) They can easily build services upon the semantically exposed data sets by expressing their functionality as semantic definitions, i.e., axioms and rules. A semantic reasoner can then derive new knowledge through definitions out of the incoming data. As such, the functionality of each service is semantically clear. (ii) Additional personalized, local semantic services can be built that intelligently and efficiently filter the high-volume and high-velocity sensor data to only forward relevant data to the semantic reasoners according to medical domain knowledge, the patient profile, background knowledge and possible other context information.

(c) **Integrators:** By leveraging the semantic descriptions of services and other potential workflow steps, they can use reasoning to dynamically construct workflows that fulfill a particular functionality. Moreover, these descriptions can also be leveraged to ensure that particular quality constraints are met.

(d) **Installers:** By using Semantic Web technologies, installers can focus on creating the required semantic definitions, i.e., models, rules & axioms, without bothering with the technological and heterogeneous details of custom interfaces.

The remainder of the chapter is organized as follows. Section 6.2 first presents the individual building blocks that are part of the semantic tool stack. Moreover, it presents a reference architecture that shows how the different building blocks can

be put together to provision distributed, data-driven, personalized continuous homecare. Finally, it discusses the details of a demonstrator on a specific use case scenario in continuous homecare, focusing on personalized smart monitoring and cross-organizational treatment planning. Section 6.3 presents the results of a performance evaluation of the different building blocks on the use case demonstrator. Finally, Section 6.4 and Section 6.5 discuss and conclude how the different building blocks solve the presented challenges, and validate the hypotheses.

6.2 Methods

This section presents the individual building blocks of the tool tack built on Semantic Web technologies, the associated reference architecture, and the details of the use case demonstrator.

6.2.1 Building blocks

The different building blocks are split up according to the different technical roles identified in Section 6.1: data providers, service providers, integrators and installers.

6.2.1.1 Data providers: semantic exposure of high-velocity data

To generate Linked Data from heterogeneous data sources, different mapping languages exist [18]. These mapping languages can be considered as schema transformation descriptions, since they allow describing the mapping policy from source schema to target schema for the involved data sources. Existing mapping languages include the RDF Mapping Language (RML) [19], xR2RML [20], D2RML [21], Dataset Representation (D-REPR) [22], and more. In this chapter, RML is chosen as it is considered the most popular mapping language to date [18].

RML is a generic mapping language that can be used to define customized mapping rules from heterogeneous data structures and serializations to the RDF data model in a declarative way [19]. RML is defined as a superset of the RDB to RDF Mapping Language (R2RML), which is the W3C recommendation for mapping relational databases to RDF [23]. This way, the purpose of RML is to extend the applicability and broaden the scope of R2RML.

To perform the actual generation of RDF graphs from heterogeneous data sources with RML, different materialization implementations exist. Examples include RMLMapper [24], MapSDI [25], GeoTriples [26], and more [18]. RMLMapper is one of the first, well-known Java implementations to perform this task based on a set of defined declarative RML mapping rules [24]. Before the RDF generation starts, it sequentially ingests multiple data sources. During ingestion, all data is loaded in memory. Hence, the available memory resources of a set-up impose a strict limitation on the amount of data that can be ingested. Therefore, an alternative methodology

was designed to parallelize the ingestion of data sources by distributing the ingestion over multiple nodes [27]. This way, the generation of RDF tasks is scaled with the volume of the data, allowing systems to generate RDF data in larger volumes.

The alternative methodology splits up the generation of RDF data in three tasks: ingestion, mapping and combination [27]. First, the ingestion task parallelizes on multiple levels: it ingests the multiple data sources in parallel, splits the data in smaller data chunks, and deserializes the different data records from each chunk in memory. This way, data records are ingested in parallel buffers, which are consumed by the mapping task to generate RDF data based on the defined rules. Applying those rules to the data records happens in parallel as much as possible, taking into account relations between data sources. Finally, the generated RDF data in the multiple buffers is concurrently read and merged to a final RDF data source by the combination task.

The presented methodology was implemented in Scala, resulting in the RMLStreamer materialization tool [27]. RMLStreamer is built using Apache Flink, which is a distributed processing framework that can easily execute the different tasks of the RDF data generation in a parallelized way through a pipeline. The configuration of the pipeline is based on RML rules. Pipeline tasks which can be parallelized are distributed over multiple instances. The implementation uses the producer-consumer approach. In this chapter, the RMLStreamer tool is chosen as materialization implementation, since it is the only existing implementation using RML that supports the generation of RDF data from high-velocity streaming data [18].

6.2.1.2 Service providers: semantic service exposure on high-velocity data

Multiple tools allow providing semantic services on the generated high-velocity data. This section details two of them: DIVIDE and Streaming MASSIF.

DIVIDE DIVIDE [28, 29] is a semantic IoT platform component that can automatically derive queries for stream processing components in an adaptive and context-aware way. By doing so, it helps solving the challenges that currently exist in the real-time processing and reasoning on high-velocity streaming data in an IoT environment. In IoT application domains such as healthcare, information about the application context regularly changes. In the area of real-time environment monitoring, this context influences how the sensor data streams in the IoT environment are being processed by stream processing components. For example, depending on the diagnosis of a certain patient, some sensors need to be monitored more closely, while others can be ignored. In essence, this defines the queries that run on the platform's RSP components.

A wide variety of IoT platforms exist for a complex IoT domain such as healthcare [30], of which multiple ones are employing semantic technologies to address the challenge of providing applications that process the IoT data in real-time [31–34]. In such existing semantic IoT platforms, the configuration of the queries for the platform's RSP components is not yet automated, adaptive and context-aware. Instead,

this configuration is still a manual task. Therefore, stream processing components typically run fixed generic queries, that use real-time semantic reasoning on all sensor data to determine from the context and domain knowledge which sensor data is relevant, and which sensor observations should be filtered for further investigation. However, to do so, highly expressive reasoning is often required in complex IoT domains such as healthcare, causing severe performance issues with high-velocity data streams and/or when fast query evaluation is required.

DIVIDE tries to solve these issues by working with non-generic, sensor-specific queries for each RSP component, allowing these queries to be continuously evaluated without the need to perform any more reasoning. It does this by performing upfront semantic reasoning on the current environmental context within the application, in order to automatically derive and configure the queries that filter observations that are relevant given the current context and the goal of the use case. To do so, it makes use of a new formalism that allows semantically representing generic query patterns in a declarative way, which are instantiated through rule-based semantic reasoning. The reasoning for the query derivation can be performed centrally on a server, and only happens each time the application context relevant to a certain stream processing component is updated. As a consequence, the resulting RSP queries do not require real-time reasoning during their continuous evaluation. Through its design, DIVIDE can automatically adapt the configured queries upon context changes, ensuring they are contextually relevant at all times.

Streaming MASSIF Streaming MASSIF [35] is a cascading reasoning framework that enables to perform expressive semantic reasoning over high velocity streams. In this domain, there often is a mismatch between expressive reasoning and high velocity streams, as the change frequency of the streams is too high to be evaluated using highly expressive reasoning. However, this expressivity is often mandatory, to either include the domain knowledge, the domain logic or abstract the low level data details from the users and allow easy query definitions. Cascading reasoning solves this mismatch by incorporating various layers of processing with various levels of expressivity [36]. In the lowest layers, lowly expressive techniques can be directly evaluated over the highly volatile data streams. They can select, using this low expressivity techniques, those parts of the stream that might be relevant for further processing. When going up in the layers, each layer processes the selection of the previous layers, thus processing less and less data. Each layer also increases the expressivity of processing. As the expressivity rises and the size of the data decreases, it is possible to evaluate highly expressive reasoning over highly volatile data streams.

Streaming MASSIF is the first realization of the cascading reasoning vision. It employs three layers. The lowest layer is the selection layer, which efficiently selects those parts of the data stream that are relevant for further processing. C-Sprite can

be employed in the selection layer. This is a reasoning system that employs an optimized reasoning algorithm for the efficient hierarchical reasoning on high-velocity data streams [37]. Regular RSP engines can also be used in the selection layer. Examples of existing, well-known RSP engines are C-SPARQL [38], SPARQL$_{Stream}$ [39], Yasper [40] and RSP4J [41]. The selections can then be abstracted in the abstraction layer, which allows defining high-level concepts and hide the low-level data details. These abstractions can then be used to define temporal dependencies in the temporal reasoning layer. All these definitions can be easily provided in a declarative way. By employing this hierarchy of processing, Streaming MASSIF is able to perform expressive reasoning over high-velocity data streams.

The layered approach of Streaming MASSIF allows services on top of these layers to easily define the data they are interested in. This can be seen as a very expressive publish/subscribe mechanism employing highly expressive reasoning to significantly decrease the subscription complexity. In this process, both temporal and standard logics can be incorporated to infer implicit data. Since Streaming MASSIF is the first concrete realization of the cascading reasoning vision and supports instrumenting concrete services through its multiple layers, it was chosen as a building block of the solution presented in this chapter [35].

6.2.1.3 Integrators: functional semantic workflow engine

AMADEUS is an adaptive, goal-driven workflow composition and conflict-detection engine [42, 43]. It solves the issues with common workflow planning systems [44–46]. Many of these systems have a limited notion of change. When an event occurs that devalues the current plan after it has been composed by such engines, all possible steps need to be revised to fit the changed state. This is a cumbersome effort and is not sustainable in dynamic environments. Moreover, existing tools can either not provide personalized workflows or detect future conflicts between multiple workflows [42].

As a solution to these issues, this chapter uses AMADEUS as a building block. AMADEUS is state-aware: the workflow composition takes into account a semantic description of the current state or context. Thereby, it is driven by a Weighted State Transition logic: possible steps are declaratively described by the changes they will make to the state description, with a possible precondition. Hence, different step descriptions can be activated in different circumstances, for example to add additional steps to the plan, or to overrule other steps. AMADEUS composes a workflow that would bring the current state to the state described in the goal. To do this, it performs semantic reasoning on the different semantic representations of the data, which include the state, steps and domain knowledge. AMADEUS follows an agent-oriented decentralized Web architecture. In this architecture, the agent automates the interaction between the workflow engine, the data sources and the applications.

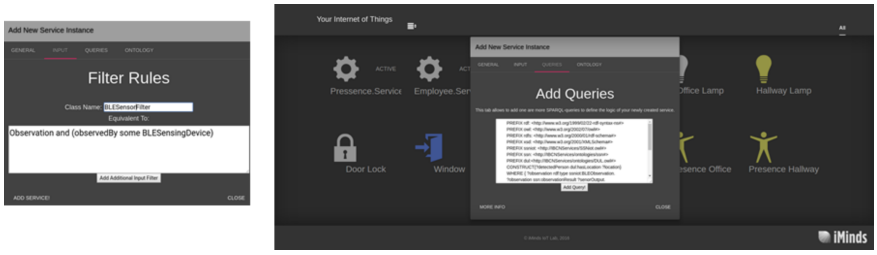


Figure 6.2: UI to visually define service subscriptions in Streaming MASSIF [49]

When employed for a specific use case, AMADEUS produces workflows that adhere to the quality constraints set up by the use case. The semantic state description reflects all what is known when the composition is performed and is iteratively modified by every step taken. The constraints are captured in the step descriptions. Events produced by services, for example via Streaming MASSIF, can make external additions or alterations to the state description and trigger a recomposition to adhere to new constraints in a new workflow. In addition, AMADEUS is able to detect possible conflicts between different workflows, for instance between the current and newly adopted workflow. This conflict detection can be applied to find future issues when the current workflow is continued. For example, when adopting AMADEUS to construct medical treatment plans, detected conflicts can be specific risks or contraindications imposed by certain combinations of treatments.

AMADEUS is implemented in the rule-based Notation3 (N3) Logic [47], which is a superset of RDF/Turtle. To compose the workflows, AMADEUS uses the EYE reasoner [48]. Its implementation contains a Web API that can be used for specific applications.

6.2.1.4 Installers: intuitive user interfaces

Multiple intuitive user interfaces (UIs) are available for installers. These include both a UI for Streaming MASSIF and graphical tools to define RML mapping rules.

Streaming MASSIF UI To simplify the service subscription in Streaming MASSIF, a query language has been developed that unifies the various layers of Streaming MASSIF. Furthermore, as shown in Figure 6.2, a UI is provided to visually define these service subscriptions [49].

Graphical tools to define RML mapping rules The RMLEditor [50] is a graph-based visualization tool to facilitate the editing of RML mapping rules that define how Linked Data is generated from various heterogeneous data sources. Using the RMLEditor, installers can create and edit declarative mapping rules, and preview the RDF data that is generated from them. As such, it is always possible for installers with

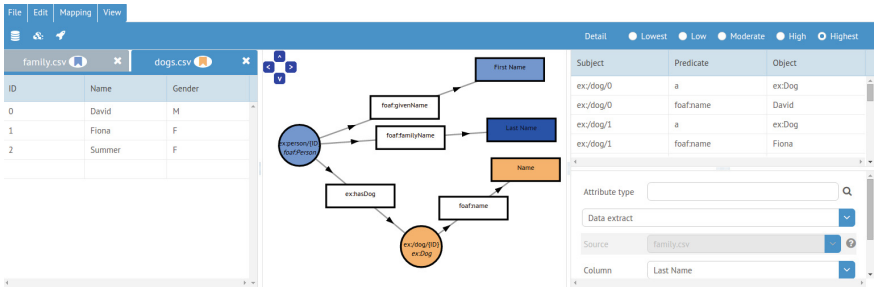


Figure 6.3: GUI of the RMLEditor [50]

sufficient domain knowledge to generate Linked Data, even if they do not have knowledge about the Semantic Web in general or the used mapping language in particular.

The RMLEditor uses a visual notation for mapping rules called MapVOWL [51]. Its architectural design consists of three layers: a presentation layer, an application layer and a data access layer. The purpose of these layers is to separate the presentation, the actual logic of the mapping process, and the access to the different input data sources. The presentation layer represents the RMLEditor's graphical user interface (GUI). The task of the application layer is to process the installer's interactions with the panels of this GUI. The data access layer is responsible for handling the different input data sources and ontologies that the installer needs to define the relevant mapping rules. The Linked Data can be generated by letting the RMLEditor execute the defined mapping rules. Alternatively, these rules can also be exported.

The GUI of the RMLEditor consists of three panels that are used by installers to define the mapping rules. The Input Panel handles the input data sources by displaying their structure and raw data. The Modeling Panel shows the actual mappings in a graph-based visualization, where the color of nodes and edges matches the color given to the data source from which data is extracted to form the RDF term. Different ontologies and vocabularies can be used to define semantic annotations. The Linked Open Vocabularies (LOV) [52] are integrated to discover relevant classes, properties and datatypes. Finally, the Results Panel shows the resulting RDF triples of executing the modeled mapping rules on the input data. A screenshot of the RMLEditor's GUI is shown in Figure 6.3 [50].

Matey [53] is another tool that can be used to view and define Linked Data generation rules. It works with YARRRML [54], which is designed as a human-readable, text-based representation language for RML mapping rules. YARRRML is a subset of the YAML data serialization language [55]. Matey works as a browser-based tool. To this end, its GUI contains multiple panels. These show a sample of the input data sources, a YARRRML representation of the mapping rules which can be edited, the Linked Data resulting from applying the current mapping rules on the data sample, and the exportable and machine-processable RML rules that correspond with

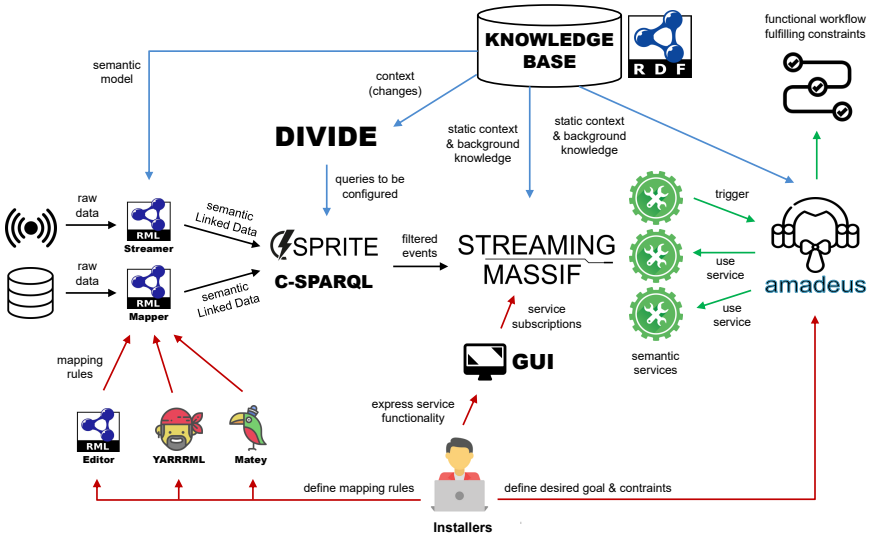


Figure 6.4: Reference architecture bringing together the different building blocks built on Semantic Web technologies. This allows optimizing continuous homecare provisioning through distributed, data-driven semantic services and cross-organizational semantic workflows.

the YARRRML representation. This way, installers can define RML mapping rules using the human-readable YARRRML representation, without requiring knowledge about the underlying mapping language.

Both the RMLEditor with MapVOWL and Matey with YARRRML help installers in generating declarative mapping rules. The RMLEditor is more adequate for data owners who are not developers, whereas Matey is more adequate for developers who are not Semantic Web experts.

6.2.2 Reference architecture

Figure 6.4 shows a reference architecture of how the different building blocks presented in the previous section can be chained together to deliver data-driven, personalized continuous homecare.

Installers should define the mapping rules to generate Linked Data through either the RMLEditor, or using the YARRRML representation language with Matey. The RML mapping rules generated by these tools are then used by the RMLStreamer (or possibly RMLMapper or other RML materialization implementations for data of low velocity) to automatically convert the incoming data (streams) to enriched Linked Data.

The selection layer of Streaming MASSIF can be represented by C-Sprite, C-SPARQL or another RSP engine. The queries that are continuously evaluated on

the chosen engine are derived and configured by DIVIDE. DIVIDE is configured with a set of generic DIVIDE query templates that define for the given use case how to intelligently filter the relevant information from the incoming streams for the various services. To derive the actual RSP queries, DIVIDE performs semantic reasoning on the relevant context in the Knowledge Base. This includes relevant domain knowledge, background knowledge and environmental context information such as the profile of the patient. By monitoring any changes to this data in the Knowledge Base, DIVIDE ensures that the correct, contextually relevant semantic stream processing queries are evaluated at all times.

The queries configured by DIVIDE on C-Sprite, C-SPARQL or similar continuously filter the Linked Data delivered by the RMLStreamer. This happens in a performant manner, since he queries are evaluated only on the data streams, and no real-time reasoning is performed during the evaluation. The filtered events are forwarded to Streaming MASSIF. Through the GUI provided by Streaming MASSIF, installers can configure the required services, by expressing their functionality as semantic queries and rules. Streaming MASSIF then performs the necessary semantic reasoning very efficiently on the incoming filtered events to deliver the desired functionality.

The services can trigger a workflow. An example is a service that raises an alarm, which triggers a workflow to select a caregiver to handle this alarm. However, an installer can also express desired functionality by semantically specifying a goal and the constraints that should be met. AMADEUS takes this specific goal as an input, which triggers the automatic construction of a workflow that fulfills this functionality according to the specified quality constraints.

6.2.3 Use case demonstrator

A demonstrator was built to showcase how the different semantic building blocks can be used to optimize continuous homecare provisioning [56]. The demonstrator is implemented on a specific use case scenario in continuous homecare, focusing on personalized smart monitoring on the sensor data streams in the patient's environment and the construction and cross-organizational coordination of patients' treatment plans.

6.2.3.1 Use case description

The scenario of the demonstrator tells the story of a patient Rosa. Rosa is an elderly woman of 74 years old that lives in a service flat in Ghent, Belgium. To follow up on Rosa, her service flat is equipped with several environmental sensors such as a light intensity sensor, a sound sensor, a room temperature sensor and a humidity sensor. Door sensors measure for every door whether it is currently open or closed. Moreover, Rosa is wearing a wearable that continuously measures her steps, body temperature and heart rate. Through multiple Bluetooth Low Energy (BLE) beacon sensors

and a BLE tag integrated into her wearable, Rosa's presence in the different rooms of the service flat can be detected. In addition, a PAS is integrated into Rosa's wearable.

According to Rosa's medical profile, she has been diagnosed with early stage dementia. Multiple people are part of Rosa's caregiver network. Nurse Suzy visits Rosa once every day in the afternoon, to assist with daily care. Dr. Wilson is Rosa's GP. Rosa is also a known patient in a nearby hospital. Moreover, two people are officially registered as informal caregivers of Rosa: her daughter Holly, who works nearby and pays Rosa a daily visit around noon, and a neighbor Roger.

6.2.3.2 Demonstrator architecture

To monitor Rosa's condition in real-time, the reference architecture in Figure 6.4 is mapped to the specific demonstrator architecture depicted in Figure 6.5. The data processing pipeline consists of the RMLStreamer, C-SPARQL and Streaming MASSIF components. C-SPARQL was chosen as RSP engine as it is one of the most well-known existing RSP engines [15, 16]. Moreover, AMADEUS is deployed as semantic workflow engine. UI components are omitted from the demonstrator architecture.

The distributed architecture contains local and central components. RMLStreamer and C-SPARQL are local components that are deployed in the patient's service flat, for example on an existing low-end local gateway device. They operate only for the patient living in that particular service flat. Streaming MASSIF, DIVIDE and AMADEUS are all central components that run on a back-end server. This could be in a server environment of either a nursing home or hospital. The central components perform their different tasks for all patients registered in the system.

In the data-driven smart monitoring pipeline, RMLStreamer maps the raw sensor data observations in JSON syntax to semantically annotated RDF observations. C-SPARQL filters the relevant RDF observations according to Rosa's profile. DIVIDE derives the correct C-SPARQL queries that perform this filtering on the local device. It does this by performing semantic reasoning on the medical domain knowledge and all contextual information in the knowledge base related to Rosa, including her medical profile. In this use case, the different diseases Rosa is diagnosed with determine these C-SPARQL queries. Streaming MASSIF performs further abstraction and temporal reasoning to infer the severity and urgency of the events filtered by C-SPARQL. It implements a service that can derive when alarming situations occur, and that can generate notifications about these alarming events to the most appropriate person in the patient's caregiver network. To decide who is the most appropriate, Streaming MASSIF takes into account the inferred event parameters and profile information such as already planned visits of caregivers.

In the described use case, AMADEUS is employed to compose semantic workflows representing a treatment plan to a disease or diagnosis. It can compose and propose different treatment plans for a diagnosis in Rosa's medical profile, and provide composed quality parameters for the treatment plan that can help the human doctor

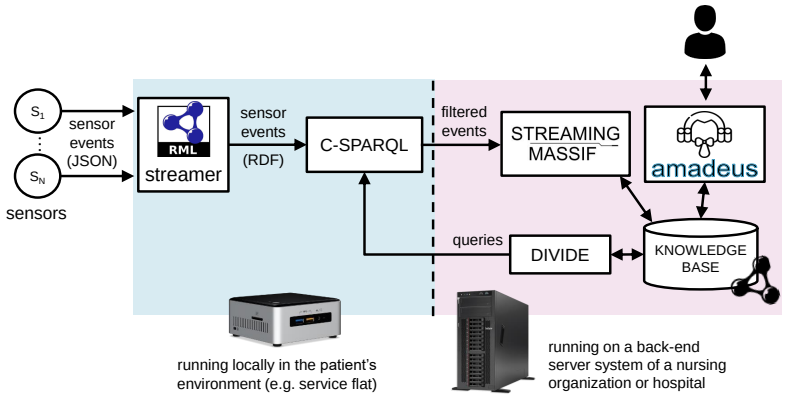


Figure 6.5: Architecture of the use case demonstrator

to select the most optimal one. Quality constraints can be defined for the proposed plans on multiple parameters such as cost, probability of success, relapse risk, patient comfort, and such. To compose the possible treatment plans, AMADEUS performs semantic reasoning using the semantic descriptions of different inputs. These include the patient's profile and medical domain knowledge about the possible options in the treatment of different diseases. These options are defined by their input, output, functionality and quality parameters. In addition, AMADEUS can also perform automatic conflict detection between treatment plans that are already activated by a doctor and a new treatment plan that is about to be added. This way, it can help a doctor in avoiding that certain conflicts are generated that the doctor is not aware of.

6.2.3.3 Scenario description

To demonstrate how the different building blocks of the demonstrator architecture work together in the presented use case, a specific scenario is designed. This scenario consists of multiple steps.

Step 0 – Initial state In its initial state, the smart monitoring pipeline is not yet activated. This means that no specific queries are evaluated on C-SPARQL. Instead, naive monitoring takes places where all raw sensor data observations are forwarded to the central server.

Step 1 – Activating the smart monitoring pipeline When the smart monitoring pipeline is activated, DIVIDE derives the personalized queries to be evaluated on the local C-SPARQL engine. Based on the generic query patterns defined within DIVIDE and Rosa's profile containing a diagnosis with early stage dementia, two queries are derived.

The first query filters observations indicating that Rosa is longer than 30 minutes in the bathroom of her service flat, without performing any movement. This query is derived because this particular scenario might indicate that an accident has happened, e.g., Rosa has fallen. Because Rosa has dementia, there is a higher chance that she might forget to use her PAS in that case. To detect this scenario, the query uses the BLE sensor in the bathroom and the wearable's step detector.

The second query filters sensor observations which imply that Rosa has left her service flat. To detect this, the BLE sensor in the hallway and the main door's sensor are monitored. Because Rosa has dementia, it is important to detect this event and notify a caregiver, since being outside alone could possibly lead to a disorientation.

Step 2 – Colon cancer diagnosis At a certain moment in time, Rosa is diagnosed with colon cancer. This cancer is diagnosed by a medical specialist at the hospital, who examined Rosa after she complained to the nurse about pain in the stomach and intestines. As a consequence, this diagnosis is added to Rosa's medical profile.

The update of Rosa's profile triggers DIVIDE to reevaluate the deployed C-SPARQL queries in a new semantic reasoning step. As a result, one additional query is derived and configured on C-SPARQL. This query detects when Rosa's body temperature exceeds 38°C (38 degrees Celsius), i.e., when Rosa has a fever. This can be detected by monitoring the sensor in Rosa's wearable. The colon cancer diagnosis leads to this new query because the medical domain knowledge states that no complications or additional infections may occur during cancer treatment. If these do occur, they form a contraindication for several cancer treatments such as chemoradiotherapy, which means that continuing these treatments would be too dangerous [57]. Since fever might indicate an underlying infection, the medical domain knowledge therefore defines that cancer patients should be monitored for fever.

Step 3 – Constructing a treatment plan for colon cancer To construct a treatment plan for Rosa's colon cancer, AMADEUS is triggered by the hospital doctor. First, AMADEUS constructs the different possible treatment plans to treat the current disease. Given Rosa's profile, the defined treatments and their quality parameters, two possible workflows are composed: a plan consisting of neoadjuvant chemoradiotherapy followed by surgery, and a plan consisting of surgery only. AMADEUS presents the different quality parameters for both options, which include duration, cost, comfort, survival rate and relapse risk. Since the first plan has the highest survival rate and lowest relapse risk, it is selected by the doctor. This selection triggers AMADEUS to perform a second reasoning run, which calculates a detailed workflow by adding timestamps to the different steps in the plan. In this case, the chemoradiotherapy step is split into four episodes of chemoradiotherapy in the hospital, with 30 days between each session. Every new session can only be performed if there is no contraindication. After confirmation of the plan, the chosen treatment plan is added

to Rosa's current treatment plan. Since the existing treatment plan of Rosa was still empty, AMADEUS does not need to perform a verification step to check whether the newly added treatment plan yields any conflicts.

Step 4 – Influenza infection yielding fever notifications Five days before her next chemoradiotherapy session, Rosa gets infected with the influenza virus. This is not an unlikely event to happen, even if she had been previously vaccinated. As a consequence, Rosa's body temperature starts to rise. When Rosa's body temperature would exceed the fever threshold of 38°C, this sensor observation would be filtered by the deployed C-SPARQL query and sent to Streaming MASSIF.

The abstraction layer of Streaming MASSIF is configured to abstract the incoming sensor events according to the following rules:

- Body temperature between 38.0°C and 38.5°C: *low fever event*
- Body temperature between 38.5°C and 39.0°C: *medium fever event*
- Body temperature above 39.0°C: *high fever event*

In addition, its temporal reasoning layer defines a *rising fever event* as a sequence of low, medium and high fever events within a time period of one hour.

Two queries are defined for the notification service instructed on top of Streaming MASSIF's temporal reasoning layer. These queries semantically represent the following rules:

- When a *low fever event* is detected, and a person in the patient's caregiver network has already planned a visit to the patient on the current day, this person should be notified to check up on the patient during this visit. In that case, no other (medical) caregiver should be called.
- When a *rising fever event* is detected, a medical caregiver from the patient's caregiver network should be notified as quickly as possible.

In the use case scenario, in the morning of the given day, Rosa's body temperature exceeds the fever threshold of 38°C. Hence, Rosa's body temperature event will be filtered by C-SPARQL, and classified by Streaming MASSIF as a *low fever event*. In the current use case, the daily visit of Rosa's daughter Holly around noon is still planned for the current day. Therefore, a notification to Holly is generated by Streaming MASSIF, indicating that Holly should check up on Rosa's low fever during her planned visit.

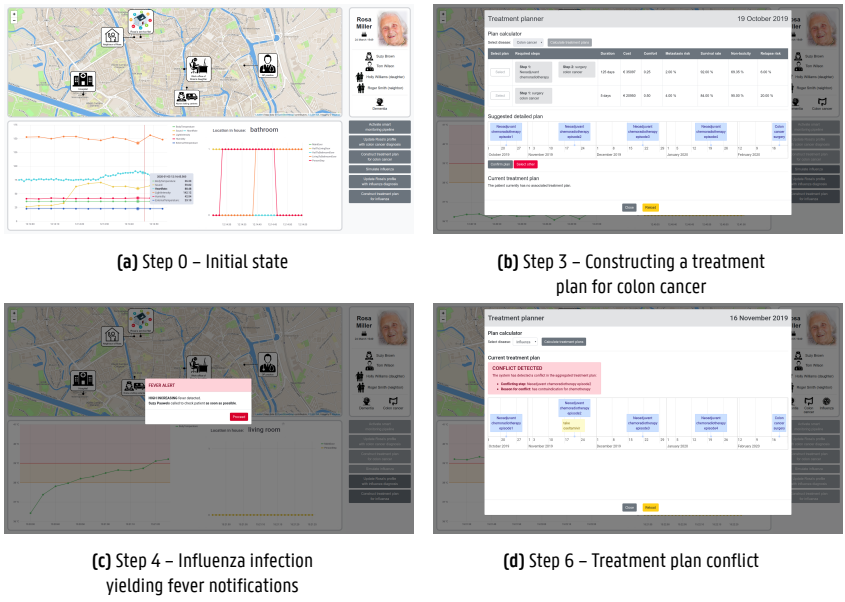
Within an hour after the first low fever event, Rosa's body temperature further rises to above 39°C. As a consequence, Streaming MASSIF detects and generates both a *medium fever event* and a *high fever event* in its abstraction layer, and thus a corresponding *rising fever event* in its temporal reasoning layer. Hence, the Streaming MASSIF service generates a notification to Rosa's nurse Suzy to visit her with high priority.

Step 5 – Constructing a treatment plan for influenza After the visit of Rosa's nurse Suzy, she decides that Rosa's GP should further examine Rosa. After examining Rosa, dr. Wilson diagnoses her with the influenza virus. This new diagnosis is also added to Rosa's medical profile. To construct a treatment plan for Rosa's influenza, dr. Wilson can use AMADEUS. After its first reasoning step, AMADEUS proposes three possible treatment plans: taking the oseltamivir medicine for ten days, taking the zanamivir medicine for eight days, or waiting for 16 days until the influenza goes over naturally. The durations of the treatment plans resemble the expected time after which the influenza virus should be cured. Given Rosa's situation, dr. Wilson decides to choose the first plan, which has the highest value for the comfort quality parameter. After selecting the plan, AMADEUS constructs the detailed workflow, which consists of taking the medication every day for a period of ten days.

Step 6 – Treatment plan conflict Before AMADEUS adds the detailed treatment plan confirmed by dr. Wilson to Rosa's current treatment plan, it performs a verification step to ensure that the newly added treatment plan does not yield any conflicts with the currently existing treatment plan. In this scenario, a conflict is detected: Rosa's next chemoradiotherapy session in the colon cancer treatment plan is scheduled in five days, while the influenza treatment plan still takes ten days. This implies that the influenza virus will not be cured on the scheduled chemoradiotherapy session, which forms a contraindication. Hence, this contraindication conflict is reported by AMADEUS. AMADEUS does not resolve detected conflicts itself, but leaves this to its end users. In this case, dr. Wilson can manually solve the conflict by postponing the next chemoradiotherapy session until the influenza virus is fully cured.

6.2.3.4 Demonstrator web application

To visually demonstrate the described use case scenario, a web application was designed [56]. This web application is built on top of a Proof-of-Concept (PoC) implementation of the use case demonstrator. Details of this implementation are presented in the next subsection. The web application illustrates how medical care providers could be able to follow up patients in homecare through the smart monitoring pipeline, in addition to the designed GUIs for the semantic tools presented in Section 6.2.1. More specifically, it simulates different aspects and shows a visualization of this simulation. This is done for Rosa's profile, the location of Rosa and the people in her caregiver network, and the real-time observations generated by the different sensors that are being processed by the monitoring pipeline. Furthermore, the web application contains a UI to trigger AMADEUS and visualize its output. Through multiple UI buttons, the web application allows browsing through the different steps of the demonstrator scenario. Figure 6.6 shows multiple screenshots of the web application corresponding to the different steps of the use case scenario. Moreover, a video of the demonstrator is available online at https://vimeo.com/380716692.



(a) Step 0 – Initial state

(b) Step 3 – Constructing a treatment plan for colon cancer

(c) Step 4 – Influenza infection yielding fever notifications

(d) Step 6 – Treatment plan conflict

Figure 6.6: Screenshots of the web application built on top of the use case demonstrator's PoC implementation. The screenshots correspond to different steps in the use case scenario.

6.2.3.5 Implementation details

This section discusses the PoC implementation of the use case demonstrator. It provides details about the configuration of the different building blocks in the demonstrator architecture in Figure 6.5. In the implementation, the domain knowledge, context information of Rosa and sensor observations are semantically annotated using an extended version of the ACCIO continuous care ontology [58, 59].

RMLStreamer The RMLStreamer maps each observation in the JSON input stream to an observation in the RDF output stream. To this end, concepts and relations defined in the extended version of the ACCIO ontology are used. An example JSON observation is shown in Listing 6.1. The resulting RDF observation after mapping it with RML mapping rules is shown in Listing 6.2.

To define how the mapping should be performed by the RMLStreamer, an RML mapping file needs to be configured. The RML mapping file used in the PoC implementation of the use case demonstrator is presented in Listing 6.3. The last part of this mapping file (lines 73–77) defines the input stream for the RMLStreamer: when started, a job is started on an Apache Flink cluster which opens a connection to a TCP socket stream on a certain host and port to pull the incoming messages from the stream. The semantically annotated sensor observations in RDF are then pushed by this job on a TCP socket output stream on a user-defined port.

Listing 6.1: Example JSON input data file that can be mapped to the RDF data in Listing 6.2 using the RML mapping file in Listing 6.3

```
{
  "observations": [
    {
      "id": "123e4567-e89b-12d3-a456-556642440000",
      "observedProperty": "PersonStep",
      "madeBySensor": "c1-19-24-70-fb-6d-S2",
      "time": "2023-04-17T14:48:22.850Z",
      "value": 1
    }
  ]
}
```

Listing 6.2: Example sensor observation in RDF/Turtle syntax, represented in the ACCIO continuous care ontology, which is the result of mapping the example JSON input data file in Listing 6.1 using the RML mapping file in Listing 6.3

```
@prefix entity: <http://occs.intec.ugent.be/ontology/entity#> .
@prefix obs: <http://occs.intec.ugent.be/ontology/observations#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .
@prefix General: <http://IBCNServices.github.io/Accio-Ontology/General.owl#> .
@prefix SSNiot: <http://IBCNServices.github.io/Accio-Ontology/SSNiot.owl#> .
@prefix DUL: <http://IBCNServices.github.io/Accio-Ontology/ontologies/DUL.owl#> .

obs:Observation_123e4567-e89b-12d3-a456-556642440000
    rdf:type sosa:Observation ;
    General:hasId [ General:hasID "123e4567-e89b-12d3-a456-556642440000"^^xsd:string ] ;
    sosa:observedProperty [ rdf:type SSNiot:PersonStep ] ;
    sosa:madeBySensor entity:c1-19-24-70-fb-6d-S2 ;
    sosa:resultTime "2023-04-17T14:48:22.850Z"^^xsd:dateTime ;
    sosa:hasResult [
        DUL:hasDataValue "1"^^xsd:float ;
    ] .
```

Listing 6.3: RML mapping file used by the RMLStreamer in the PoC implementation of the use case demonstrator, presented in RDF/Turtle syntax (part 1/2). `[...]` is a placeholder for omitted parts that are not of interest.

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix rmls: <http://semweb.mmlab.be/ns/rmls#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix map: <http://mapping.example.com/>.

# defines which triple subjects should be linked to the different predicate-object maps
map:map_observations_0 rml:logicalSource map:source; a rr:TriplesMap;
    rdfs:label "observations"; rr:subjectMap map:s_0;
    rr:predicateObjectMap map:pom_0, map:pom_1, map:pom_2, map:pom_3,
                          map:pom_4, map:pom_5.
map:map_ids_0 rml:logicalSource map:source; a rr:TriplesMap;
    rdfs:label "ids"; rr:subjectMap map:s_1; rr:predicateObjectMap map:pom_6.
map:map_props_0 rml:logicalSource map:source; a rr:TriplesMap;
    rdfs:label "props"; rr:subjectMap map:s_2; rr:predicateObjectMap map:pom_7.
map:map_results_0 rml:logicalSource map:source; a rr:TriplesMap;
    rdfs:label "results"; rr:subjectMap map:s_3; rr:predicateObjectMap map:pom_8.

# define the object maps (which entities and datatypes should be present in
#                         the objects of the 9 resulting RDF triples)
map:om_0 a rr:ObjectMap; rr:termType rr:IRI;
    rr:constant "http://www.w3.org/ns/sosa/Observation".
map:om_1 a rr:ObjectMap; rr:termType rr:Literal;
    rr:template "http://occs.intec.ugent.be/ontology/observations#Observation_{id}_id".
map:om_2 a rr:ObjectMap; rr:termType rr:Literal;
    rr:template "http://occs.intec.ugent.be/ontology/observations#Observation_{id}_prop".
map:om_3 a rr:ObjectMap; rr:termType rr:Literal;
    rr:template "http://occs.intec.ugent.be/ontology/entity#{madeBySensor}".
map:om_4 a rr:ObjectMap; rml:reference "time"; rr:termType rr:Literal;
    rr:datatype <http://www.w3.org/2001/XMLSchema#datetime>.
map:om_5 a rr:ObjectMap; rr:termType rr:Literal; rr:template
    "http://occs.intec.ugent.be/ontology/observations#Observation_{id}_result".
map:om_6 a rr:ObjectMap; rml:reference "id"; rr:termType rr:Literal;
    rr:datatype <http://www.w3.org/2001/XMLSchema#string>.
map:om_7 a rr:ObjectMap; rr:termType rr:IRI; rr:template
    "http://IBCNServices.github.io/Accio-Ontology/SSNiot.owl#{observedProperty}".
map:om_8 a rr:ObjectMap; rml:reference "value"; rr:termType rr:Literal;
    rr:datatype <http://www.w3.org/2001/XMLSchema#double>.

# define the predicate maps
# (which predicates should be used in the 9 resulting RDF triples)
map:pm_0 a rr:PredicateMap; rr:constant rdf:type.
map:pm_1 a rr:PredicateMap;
    rr:constant <http://IBCNServices.github.io/Accio-Ontology/General.owl#hasId>.
map:pm_2 a rr:PredicateMap; rr:constant <http://www.w3.org/ns/sosa/observedProperty>.
map:pm_3 a rr:PredicateMap; rr:constant <http://www.w3.org/ns/sosa/madeBySensor>.
map:pm_4 a rr:PredicateMap; rr:constant <http://www.w3.org/ns/sosa/resultTime>.
map:pm_5 a rr:PredicateMap; rr:constant <http://www.w3.org/ns/sosa/hasResult>.
map:pm_6 a rr:PredicateMap;
    rr:constant <http://IBCNServices.github.io/Accio-Ontology/General.owl#hasID>.
map:pm_7 a rr:PredicateMap; rr:constant rdf:type.
map:pm_8 a rr:PredicateMap;
    rr:constant <http://IBCNServices.github.io/Accio-Ontology/ontologies/DUL.owl#
          hasDataValue>.

# link the predicates to the objects in a predicate-object map
map:pom_0 a rr:PredicateObjectMap; rr:predicateMap map:pm_0; rr:objectMap map:om_0.
[...]
map:pom_8 a rr:PredicateObjectMap; rr:predicateMap map:pm_8; rr:objectMap map:om_8.
```

Listing 6.3: RML mapping file used by the RMLStreamer in the PoC implementation of the use case demonstrator, presented in RDF/Turtle syntax (part 2/2)

```
61  # define subject maps (containing templates representing which entities should be
62  #                       used in the subjects of the 9 resulting triples)
63  map:s_0 a rr:SubjectMap; rr:template
64      "http://occs.intec.ugent.be/ontology/observations#Observation_{id}".
65  map:s_1 a rr:SubjectMap; rr:template
66      "http://occs.intec.ugent.be/ontology/observations#Observation_{id}_id".
67  map:s_2 a rr:SubjectMap; rr:template
68      "http://occs.intec.ugent.be/ontology/observations#Observation_{id}_prop".
69  map:s_3 a rr:SubjectMap; rr:template
70      "http://occs.intec.ugent.be/ontology/observations#Observation_{id}_result".
71
72  # define source (input data stream) for RMLStreamer to read from
73  map:source a rml:LogicalSource;
74      rml:source [ rdf:type rmls:TCPSocketStream ;
75                   rmls:hostName "192.168.1.49";
76                   rmls:type "PULL" ; rmls:port "5005" ];
77      rml:referenceFormulation ql:JSONPath.
```

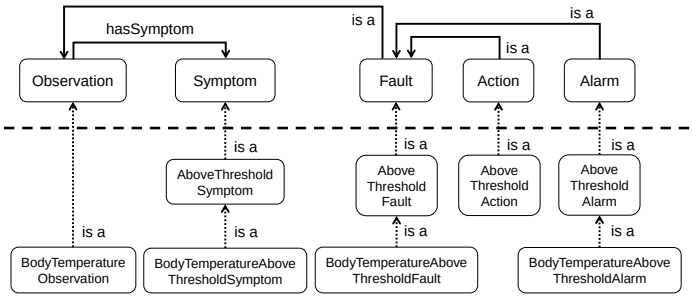


Figure 6.7: Observation pattern in the ACCIO continuous care ontology. This pattern is used in the use case demonstrator in the DIVIDE query that yields the C-SPARQL query filtering high body temperature (fever) events.

DIVIDE and C-SPARQL The contextually relevant C-SPARQL queries are derived and configured by DIVIDE when the use case context associated to patient Rosa is updated. To derive specific queries in DIVIDE, generic versions of these queries need to be loaded into the system. Based on the context and domain knowledge, the DIVIDE query derivation can then perform reasoning to derive for which sensors these queries need to be instantiated, and how this instantiation should happen.

The ACCIO ontology makes use of an observation pattern involving the classes `Observation`, `Symptom`, `Fault`, `Action` and `Alarm`. Figure 6.7 details how these classes are linked. Moreover, it gives an example of a series of subclasses that model a symptom, fault, action and alarm related to an observation of the `BodyTemperature` property that has a value exceeding a certain, medically defined threshold. This is especially relevant to understand how the query is derived that filters Rosa's fever events when her medical profile is updated with the colon cancer diagnosis.

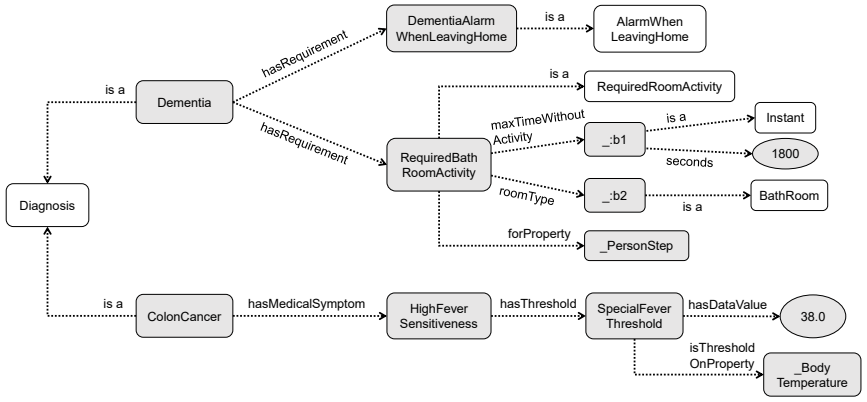


Figure 6.8: Overview of how diagnoses are modeled in the medical domain knowledge. This is shown for the diagnoses occurring in the demonstrator's use case scenario. The modeling is performed using the extended version of the ACCIO ontology. For readability purposes, ontology prefixes are omitted.

Figure 6.8 details how the diagnoses occurring in the use case scenario, dementia and colon cancer, are semantically modeled in the extension of the ACCIO continuous care ontology, according to the medical knowledge owned by the hospital about these diseases. These definitions are used by DIVIDE to convert the generic DIVIDE queries to the specific C-SPARQL queries of the use case scenario. Three generic DIVIDE queries are configured in the PoC implementation of the demonstrator. This follows from the diagnosis overview in Figure 6.8: there is one DIVIDE query corresponding to the medical symptom associated with colon cancer, and one DIVIDE query corresponding to each requirement associated to the dementia diagnosis.

Listing 6.4 presents the sensor query rule with generic query pattern of the first DIVIDE query that filters `AboveThresholdAlarm` instances. This is a subclass of the `Alarm` class of the observation pattern of the ACCIO ontology, as shown in Figure 6.7. For an `Observation` to also be an `AboveThresholdAlarm`, some conditions must be fulfilled. One of these conditions is that the `Observation` is linked to an `AboveThresholdSymptom`. The sensor query rule in Listing 6.4 links an `Observation` to an `AboveThresholdSymptom` if a threshold `?threshold` is crossed. Through reasoning, DIVIDE will only instantiate this rule, and hence the generic query pattern, for the cases where the `Observation` with an `AboveThresholdSymptom` is also an instance of `AboveThresholdAlarm`. This happens through the semantic reasoning. In Rosa's case, this query will instantiate for `?prop` being `SSNiot:BodyTemperature` when her medical profile contains the triple:

```
:Rosa CareRoomMonitoring:hasDiagnosis CareRoomMonitoring:ColonCancer .
```

The other two DIVIDE queries corresponding to the dementia diagnosis are similar to the presented DIVIDE query. However, they do not use the generic ontology

observation pattern, but directly model the requirements associated to the dementia diagnosis shown in Figure 6.8.

Streaming MASSIF In the PoC implementation of the use case demonstrator, the different layers of Streaming MASSIF are employed. The configuration details of each layer are discussed below.

- The selection layer uses the C-SPARQL engine as described before, which continuously evaluates the queries derived by DIVIDE.

- The abstraction layer of Streaming MASSIF abstracts the body temperature sensor events filtered by C-SPARQL to high-level events. This is done through expressive semantic reasoning, using the rules explained in the description of step 4 of the demonstrator's use case scenario. As an example, consider the following high-level definition to describe a *medium fever event*:

```
MediumTemperatureEvent =
    AboveTemperatureThresholdAlarm and
    hasResult some (
        (hasDataValue some xsd:double[>= "38.5"^^xsd:double]) and
        (hasDataValue some xsd:double[< "39"^^xsd:double]))
```

  The definitions to describe a *low fever event* and *high fever event* are completely similar.

- Streaming MASSIF's temporal reasoning layer detects temporal dependencies between high-level events. The high-level definition of a *rising fever event* is semantically described as follows:

```
RisingTemperatureEvent =
    every (a=LowTemperatureEvent -> b=MediumTemperatureEvent
           -> c=HighTemperatureEvent) where timer:within(3600 sec)
```

On top of Streaming MASSIF's temporal reasoning layer, two queries are defined for the instructed notification service. The query representing the rule to send a caregiver with a scheduled visit to the patient in case of a *low fever event*, is shown in Listing 6.5. The other query processing any *rising fever event* is very similar.

AMADEUS To compose a workflow representing a treatment plan to Rosa's colon cancer, AMADEUS starts from the current state defined in the knowledge base. In the demonstrator use case, this RDF state description contains Rosa's personal information, and medical information: diagnosis, tumor size, risk of metastasis, etc. An example of this state description is shown in Listing 6.6. To represent the medical diagnoses, the implementation makes use of the Systematized Nomenclature of Medicine Clinical Terms (SNOMED CT) [60].

Listing 6.4: Sensor query rule with the generic query pattern of the DIVIDE query in the use case demonstrator that filters instances of the `AboveThresholdAlarm` class (part 1/2)

```
@prefix : <http://idlab.ugent.be/sensdesc/query#> .
@prefix sd: <http://idlab.ugent.be/sensdesc#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ssn: <http://www.w3.org/ns/ssn/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .
@prefix DUL: <http://IBCNServices.github.io/Accio-Ontology/ontologies/DUL.owl#> .
@prefix SSNiot: <http://IBCNServices.github.io/Accio-Ontology/SSNiot.owl#> .
@prefix RoleCompetenceAccio:
    <http://IBCNServices.github.io/Accio-Ontology/RoleCompetenceAccio.owl#> .
@prefix CareRoomMonitoring:
    <http://IBCNServices.github.io/Accio-Ontology/CareRoomMonitoring.owl#> .

{
    ?p DUL:hasRole [ rdf:type RoleCompetenceAccio:PatientRole ] ;
       DUL:hasLocation ?l ;
       CareRoomMonitoring:hasDiagnosis [
           CareRoomMonitoring:hasMedicalSymptom [
               rdf:type CareRoomMonitoring:HighSensitivity ;
               SSNiot:hasThreshold [
                   DUL:hasDataValue ?threshold ;
                   SSNiot:isThresholdOnProperty [ rdf:type ?prop ]
               ]
           ]
       ] .

    ?sensor rdf:type sosa:Sensor ;
            sosa:observes [ rdf:type ?prop ] ;
            SSNiot:isSubsystemOf [ DUL:hasLocation ?l ] .

    ?prop rdfs:subClassOf sosa:ObservableProperty .
} => {
    _:q rdf:type sd:Query ;
        sd:pattern :pattern-above-threshold-alarm ;
        sd:inputVariables (("?prop" ?prop) ("?threshold" ?threshold)
                           ("?sensor" ?sensor) ("?patient" ?p)) ;
        sd:outputVariables (("?v" _:v) ("?o" _:oo)) .

    _:oo rdf:type sosa:Observation ;
         sosa:madeBySensor ?sensor ;
         sosa:hasResult [
             rdf:type SSNiot:QuantityObservationValue ;
             DUL:hasDataValue _:v ] ;
         SSNiot:hasSymptom [
             rdf:type CareRoomMonitoring:AboveThresholdSymptom ;
             ssn:forProperty [ rdf:type ?prop ] ] .
} .

:prefixes-above-threshold-alarm rdf:type owl:Ontology ;
    sh:declare [ sh:prefix "xsd" ;
                 sh:namespace "http://www.w3.org/2001/XMLSchema#"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "ssn" ; sh:namespace "http://www.w3.org/ns/ssn/"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "sosa" ;
                 sh:namespace "http://www.w3.org/ns/sosa/"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "General" ; sh:namespace "http://IBCNServices.github.io/Accio-
          Ontology/General.owl#"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "CareRoomMonitoring" ; sh:namespace "http://IBCNServices.github.io/
          Accio-Ontology/CareRoomMonitoring.owl#"^^xsd:anyURI ] ;
    sh:declare [ sh:prefix "DUL" ; sh:namespace "http://IBCNServices.github.io/Accio-Ontology/
          ontologies/DUL.owl#"^^xsd:anyURI ] .
```

Listing 6.4: Sensor query rule with the generic query pattern of the DIVIDE query in the use case demonstrator that filters instances of the `AboveThresholdAlarm` class (part 2/2)

```
:pattern-above-threshold-alarm
    rdf:type sd:QueryPattern ;
    sh:prefixes :prefixes-above-threshold-alarm ;
    sh:construct """
        CONSTRUCT {
            ?o a CareRoomMonitoring:AboveThresholdAlarm ;
               ssn:forProperty ?prop ;
               DUL:associatedWith ?patient ;
               sosa:hasResult [ DUL:hasDataValue ?v ] .
        }
        FROM NAMED WINDOW :win ON <http://idlab.ugent.be/grove> [RANGE PT5S SLIDE PT3S]
        WHERE {
            WINDOW :win {
                ?o a sosa:Observation ;
                   sosa:madeBySensor ?sensor ;
                   sosa:hasResult [ DUL:hasDataValue ?v ] ;
                   sosa:resultTime ?t ;
                   General:hasId [ General:hasID ?id ] .
                FILTER (xsd:double(?v) > xsd:double(?threshold))
            }
        }
        ORDER BY DESC(?t)
        LIMIT 1
        """ .
```

Listing 6.5: Query of Streaming MASSIF's instructed notification service that generates a notification to a caregiver with a scheduled patient visit in case of a low fever event

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX DUL: <http://IBCNServices.github.io/Accio Ontology/ontologies/DUL.owl#>
PREFIX role: <http://ibcnservices.github.io/Accio Ontology/RoleCompetenceAccio.owl#>
PREFIX : <http://idlab.dissect.healthdemo/selectionservice.owl#>

CONSTRUCT { ?visitor rdf:type :VisitorLowPriority}
WHERE {
    ?fever rdf:type :LowTemperatureEvent.
    ?fever DUL:associatedWith ?patient.
    ?patient :hasSchedule ?schedule.
    ?schedule :hasDaySchedule :todaysSchedule.
    :todaysSchedule :hasItem ?item.
    ?item :hasVisitor ?visitor.
    ?visitor DUL:hasRole ?role.
    ?role rdf:type role:Child.
    FILTER NOT EXISTS { ?rising rdf:type :RisingTemperatureEvent }
}
```

Listing 6.6: Initial state description used by AMADEUS in the use case demonstrator to compose medical treatment plans to treat colon cancer, in N3 syntax

```
PREFIX sct: <http://snomed.info/id/>
PREFIX data: <https://gitlab.ilabt.imec.be/KNoWS/dissect/data#>
PREFIX care: <https://gitlab.ilabt.imec.be/KNoWS/dissect/care#>

# Rosa's patient data
data:patient_1 a care:Patient.
data:patient_1 care:age 74 .
data:patient_1 care:name "Rosa" .
data:patient_1 care:gender "female" .
data:patient_1 care:weight 63 .

# colon cancer diagnosis
data:patient_1 care:diagnosis sct:363406005, sct:363351006 .
data:patient_1 care:tumor_size 40 .
data:patient_1 care:metastasis_risk 0.4 .
data:patient_1 care:5yr_survival_rate 0.2 .
data:patient_1 care:non_toxicity 1 .
data:patient_1 care:position sct:34402009 .
data:patient_1 care:status "active" .
data:patient_1 care:tnm_t 3 .
data:patient_1 care:blocking_colon false .
data:patient_1 care:5yr_local_relapse_risk 0 .
```

When running the EYE reasoner, a goal should be defined to represent what the target state is that EYE should be looking for when composing workflows. An example of the goal description for the demonstrator use case is shown in Listing 6.7. Moreover, the inputs of the EYE reasoner contain different policies as a Weighted Transition Logic in N3 with medical domain knowledge about treating colon cancer. These policies are essentially step descriptions, describing the changes they will make to the state description. Listing 6.8 contains the example of a colon cancer policy representing the possible surgery step in colon cancer treatment. Listing 6.9 contains some examples of additionally relevant medical domain knowledge: preconditions that allow a patient to take surgery, rules to calculate the relapse risk after surgery for different situations, and definitions of contraindications that can result in conflicting workflows.

6.3 Results

This section evaluates the performance of the different building blocks in the architecture of the use case demonstrator presented in Section 6.2.3 [56]. The evaluation is split up in three parts. The first part presents the performance evaluation of the data stream processing pipeline involving RMLStreamer, C-SPARQL and Streaming MASSIF. The second part details the evaluation of the query derivation with DIVIDE. The third part discusses the evaluation of AMADEUS.

For all evaluations, the local components in the demonstrator architecture in Figure 6.5 (RMLStreamer, C-SPARQL) are running on an Intel NUC, model

Listing 6.7: Goal description used by AMADEUS in the use case demonstrator to compose medical treatment plans to treat colon cancer, in N3 syntax

```
PREFIX math: <http://www.w3.org/2000/10/swap/math#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX gps: <http://josd.github.io/eye/reasoning/gps/gps-schema#>
PREFIX sct: <http://snomed.info/id/>
PREFIX care: <https://gitlab.ilabt.imec.be/KNoWS/dissect/care#>

{
    ?SCOPE gps:findpath (
    {
        ?patient a care:Patient.
        ?patient care:diagnosis sct:363406005.
        ?patient care:tumor_size 0 .
        ?patient care:metastasis_risk ?risk .
        ?patient care:5yr_survival_rate ?rate .
        ?patient care:non_toxicity ?non_toxicity .
        ?patient care:5yr_local_relapse_risk ?relapse_risk .

        # additional requirements for the treatment plan could be defined as shown below
        # ?risk math:lessThan 0.1 .
        # ?rate math:greaterThan 0.7 .
        # ?non_toxicity math:greaterThan 0.5 .
        # ?relapse_risk math:lessThan 0.15 .
    }
    ?PATH ?DURATION ?COST ?BELIEF ?COMFORT
    ("P150D"^^xsd:dayTimeDuration 200000.0 0.1 0.1)).
} => {
    ?patient gps:path (?PATH ?DURATION ?COST ?BELIEF ?COMFORT
                       (?risk ?rate ?non_toxicity ?relapse_risk)).
} .
```

Listing 6.8: Example of a colon cancer policy (surgery step description) used by AMADEUS in the use case demonstrator to compose medical treatment plans to treat colon cancer, in N3 syntax

```
PREFIX math: <http://www.w3.org/2000/10/swap/math#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX e: <http://eulersharp.sourceforge.net/2003/03swap/log-rules#>
PREFIX gps: <http://josd.github.io/eye/reasoning/gps/gps-schema#>
PREFIX action: <https://gitlab.ilabt.imec.be/KNoWS/dissect/action#>
PREFIX sct: <http://snomed.info/id/>
PREFIX surgery: <https://gitlab.ilabt.imec.be/KNoWS/dissect/surgery#>
PREFIX care: <https://gitlab.ilabt.imec.be/KNoWS/dissect/care#>

# surgery step description
{
    care:Colon_cancer gps:description (
        {
            ?patient care:tumor_size ?size.
            ?patient care:metastasis_risk ?risk .
            ?patient care:5yr_survival_rate ?rate .
            ?patient care:non_toxicity ?non_toxicity .
            ?patient care:5yr_local_relapse_risk ?relapse_risk .
        }
        { ?patient gps:surgery surgery:surgery_colon_cancer. }
        {
            # surgery should completely remove the tumor
            ?patient care:tumor_size 0 .
            ?patient care:metastasis_risk ?new_risk .
            ?patient care:5yr_survival_rate ?new_rate .
            ?patient care:non_toxicity ?new_non_toxicity .
            ?patient care:taken action:surgery_colon_cancer.
            ?patient care:5yr_local_relapse_risk ?new_relapse_risk .
        }
        action:surgery_colon_cancer
        # defines duration, cost, belief & comfort quality parameters of surgery
        "P5D"^^xsd:dayTimeDuration 20950 0.9 0.5
    )
} <= {
    ?patient a care:Patient.
    ?patient care:diagnosis sct:363406005.
    ?patient care:surgery_colon_cancer_precondition true .
    ?scope e:fail { ?patient care:taken action:surgery_colon_cancer. }.

    ?patient care:post_surgery_5yr_local_relapse_risk ?new_relapse_risk .

    ?patient care:metastasis_risk ?risk .
    (?risk 0.1) math:product ?new_risk.

    # surgery decreases the 5 year death rate (= 1 - 5 year survival rate) with 80%
    ?patient care:5yr_survival_rate ?rate .
    (1 ((1 ?rate)!math:difference 0.2)!math:product) math:difference ?new_rate.

    ?patient care:non_toxicity ?non_toxicity .
    (?non_toxicity 0.95) math:product ?new_non_toxicity.
}.
```

Listing 6.9: Examples of additionally relevant medical domain knowledge, used by AMADEUS in the use case demonstrator to compose medical treatment plans to treat colon cancer and automatically detect conflicts between treatment plans, in N3 syntax

```
PREFIX math: <http://www.w3.org/2000/10/swap/math#>
PREFIX action: <https://gitlab.ilabt.imec.be/KNoWS/dissect/action#>
PREFIX sct: <http://snomed.info/id/>
PREFIX therapy: <https://gitlab.ilabt.imec.be/KNoWS/dissect/therapy#>
PREFIX care: <https://gitlab.ilabt.imec.be/KNoWS/dissect/care#>
PREFIX medication: <https://gitlab.ilabt.imec.be/KNoWS/dissect/medication#>

# preconditions that allow a patient to take surgery
{
    ?patient care:surgery_colon_cancer_precondition true .
} <= {
    ?patient a care:Patient.
    ?patient care:diagnosis sct:363406005, sct:363351006.
    ?patient care:tnm_t ?t_value .
    ?t_value math:lessThan 3 .
}.
{
    ?patient care:surgery_colon_cancer_precondition true .
} <= {
    ?patient a care:Patient.
    ?patient care:diagnosis sct:363406005, sct:363351006.
    ?patient care:tnm_t ?t_value .
    ?t_value math:greaterThan 2 .
    ?patient care:taken action:Neoadjuvant_chemoradiotherapy.
}.

# rules to calculate the relapse risk after surgery for different situations
{
    ?patient care:post_surgery_5yr_local_relapse_risk 0.04 .
} <= {
    ?patient care:tnm_t 2 .
}.
{
    ?patient care:post_surgery_5yr_local_relapse_risk 0.06 .
} <= {
    ?patient care:taken action:Neoadjuvant_chemoradiotherapy.
    ?patient care:tnm_t 3 .
}.

# definitions of contraindications that can result in conflicting workflows
# -> patient has influenza
{
    ?patient therapy:hasContraindicationForChemotherapy true
} <= {
    ?patient a care:Patient.
    ?patient care:diagnosis sct:6142004. # influenza diagnosis
}.
# -> or patient takes medications that conflict with chemoradiotherapy medications
{
    ?patient therapy:hasContraindicationForChemotherapy true
} <= {
    ?patient a care:Patient.
    ?patient therapy:medication ?med.
    ?med medication:contraindication therapy:chemoradiotherapy.
}.
```

D54250WYKH, which has a 1300 MHz dual-core Intel Core i5-4250U CPU (turbo frequency 2600 MHz) and 8 GB DDR3-1600 RAM. The central components (Streaming MASSIF, DIVIDE, AMADEUS) are deployed on a virtual Ubuntu 18.04 server with a Intel Xeon E5620 2.40GHz CPU, and 12GB DDR3 1066 MHz RAM.

The results of all evaluations presented in this section are aggregated in Table 6.1. For every evaluated component, the following subsections zoom in on the details and rationales of the evaluation cases, the definitions of the measured metrics, and the details about how the multiple measures were obtained to calculate the reported averages and standard deviations.

6.3.1 Evaluation of the data stream processing pipeline

The evaluation of the data stream processing pipeline of the use case demonstrator is performed separately for the three components. This approach is chosen because C-SPARQL performs continuous time-based processing of the data on the streams using data windows, while RMLStreamer and Streaming MASSIF do event-based processing. Analyzing the components individually means that inherent networking delays are omitted.

6.3.1.1 RMLStreamer

For the RMLStreamer evaluation, the processing time is measured, which is defined as the difference between the time at which a JSON observation is sent on the TCP socket input stream of RMLStreamer, and the time at which the semantically annotated observation in RDF arrives at the client consuming the TCP socket output stream of RMLStreamer. Both the sensor simulator hosting the input TCP socket server, and the client hosting the output TCP socket server, are running on the same device as the RMLStreamer component.

In Table 6.1, the RMLStreamer performance measures are reported for three different rates of incoming observations on the RMLStreamer: 1 observation per second, 7 observations per second and 14 observations per second. The maximum tested number of 14 is chosen because the demonstrator contains 14 sensors. However, each sensor generates observations with its own periodicity, varying from 1 second between observations to 5 seconds. Hence, the number of observations per second is always upper bounded by 14, but often also lower. The reported numbers are aggregated over all observations generated during a simulation of 2 minutes.

6.3.1.2 C-SPARQL

For the C-SPARQL evaluation, the execution time is measured of the query that is filtering Rosa's body temperature after she is diagnosed with colon cancer. This is the only query that is important for the scenario of the demonstrator. The other

Table 6.1: Results of the performance evaluation of the building blocks in the use case demonstrator's architecture

Evaluated component	Measured metric	Evaluation case	Average value (ms)	Standard deviation (ms)
RMLStreamer	processing time	1 observation per second	8.1	3.1
		7 observations per second	11.8	7.1
		14 observations per second	13.5	8.3
C-SPARQL	query execution time	query filtering body temperature with 1 observation per second	12.2	3.3
		query filtering body temperature with 7 observations per second	15.2	9.1
		query filtering body temperature with 14 observations per second	26.4	23.5
Streaming MASSIF	processing time	fever event processing	1539.5	60.1
DIVIDE	processing time	query derivation	7249.5	175.8
AMADEUS	processing time	generating treatment plans for colon cancer	190.8	1.4
		generating treatment plans for influenza	88.6	1.7
		aggregating treatment plans and performing conflict detection	1335.7	3.8

two deployed C-SPARQL queries mentioned in the use case scenario are also continuously evaluating the data streams in parallel threads, but never yield any filtered event in their results during the scenario.

Similarly to the RMLStreamer evaluation, Table 6.1 reports the C-SPARQL evaluation results for three different rates of incoming RDF observations on C-SPARQL: 1 observation per second, 7 observations per second, and 14 observations per second. For C-SPARQL, this defines the number of observations in the data window, and thus the size of the data model on which the queries are evaluated. Out of these observations, always exactly 1 observation is made by the body temperature sensor. This observation always has a value higher than 38°C. Hence, this resembles the period in the demonstrator scenario when Rosa is suffering from influenza, and has a fever for a certain period. This means that the query execution times are measured for individual query executions that each yield exactly 1 result, being the most recent high body temperature observation. The query is evaluated every 3 seconds on a window containing all data stream observations of the last 5 seconds. The reported numbers are aggregated over all query executions during a simulation of 2 minutes.

Note that the evaluation results report measures about the query execution times, and not the processing times of an observation. This is because the C-SPARQL query evaluation is a continuous process with a certain frequency, and is thus not event-based. The total processing time per observation exists on a query level (i.e., per

continuous query evaluation), and consists of 2 main parts: the waiting time, and the query execution time. The waiting time is an inherent delay caused by the continuous query processing: when an observation is published by RMLStreamer on the RDF data stream registered to C-SPARQL, it is only processed as part of the data window during the next query evaluation. Hence, the worst-case waiting time is defined by the time between consecutive evaluations of the RSP query. Since the actual waiting time is inherent to the system, depends on the mutual initialization of components, and is not dependent on the query bodies and data models, it is not included in the reported results. In the use case demonstrator, the body temperature query is evaluated every 3 seconds. Hence, the waiting time is upper bounded by 3 seconds.

6.3.1.3 Streaming MASSIF

For the evaluation of Streaming MASSIF, the processing time of an incoming event is measured. This is defined as the difference between the time at which the event arrives in the Streaming MASSIF component, and the time at which the notification (to either Rosa's daughter or nurse) leaves the system. The reported numbers in the results in Table 6.1 are aggregated over all observations generated during a simulation of 3 minutes, where Rosa's body temperature is gradually increased from 38.3°C up to 39.1°C. The period between incoming events in Streaming MASSIF is 3 seconds, since the corresponding C-SPARQL query that is filtering high body temperature observations is executed every 3 seconds.

6.3.2 Evaluation of DIVIDE

The evaluation of DIVIDE measures the processing time of the query derivation on the use case context associated to Rosa. This context includes both the dementia and colon cancer diagnosis. DIVIDE performs the semantic reasoning during the query derivation in three parallel threads, where each thread is responsible for deriving the RSP queries from one of the DIVIDE queries. The output of the DIVIDE query derivation consists of the three RSP queries as described in the demonstrator's use case scenario. The processing time is measured from when the parallel reasoning processes start, until all processes have completed. All networking overhead for registering the context to DIVIDE, which triggers the query derivation, and registering the resulting queries on C-SPARQL is not included in the reported results. The numbers reported in the evaluation results in Table 6.1 are aggregated over 30 runs, excluding 3 warm-up and 2 cool-down runs.

6.3.3 Evaluation of AMADEUS

For the evaluation of AMADEUS, the processing times are measured of a request to the AMADEUS Web API for the three most important cases associated to the

demonstrator's use case scenario: (1) requesting possible treatment plans for the colon cancer diagnosis, (2) requesting possible treatment plans for the influenza diagnosis, and (3) adding the chosen influenza treatment plan to the existing treatment plan for colon cancer, including the detection of conflicts between both plans. The processing time corresponds to the response time of the AMADEUS Web API, which mainly represents the duration of the EYE reasoner process started by AMADEUS. The evaluation results in Table 6.1 are measured over 30 runs, excluding 3 warm-up and 2 cool-down runs.

6.4 Discussion

In healthcare, it is a challenge for the different stakeholders involved in the follow-up of patients to provide the best possible care for their patients. To realize this, the communication and coordination of data, services and workflows across organizations and stakeholders should be optimized. This is only possible when addressing the individual challenges imposed to the different roles that can be discerned when looking at these challenges from a technical perspective. These roles are the data providers, service providers, integrators and installers. In this section, we discuss how the presented existing building blocks built upon Semantic Web technologies (Section 6.2.1) can help solving the challenges related to every individual role, by putting it all together according to the presented reference architecture (Section 6.2.2). To do so, relevant insights from designing the use case demonstrator (Section 6.2.3) and evaluating our PoC implementation of this demonstrator (Section 6.3) are shared as well. This way, the hypotheses presented in Section 6.1.3 are validated.

6.4.1 Data providers

In existing systems, custom APIs are typically built to expose the available data to the different stakeholders involved in the system. This data often still resides in big data silos. This makes data reuse hard, while there is also no uniform way to make the meaning of the data clear. Semantic Web technologies are perfectly suited to move away from the current approach and solve this issue: they offer the tools to formally describe different heterogeneous data sources in a uniform, machine-interpretable format. Ontologies allow explicitly and formally defining the meaning of the data. By using a common format, reusing data sources defined as Linked Data across organizations and across applications becomes possible.

Exposing data from various sources as Linked Data is possible through RML mappings. RMLMapper is a tool that can process such mapping rules and generate Linked Data. RMLStreamer is another tool that solves the issue with previous tools that custom data mappings often do not scale and cannot keep up with the velocity of the

incoming data stream. It does this by parallelizing the Linked Data generation process as much as possible, and reducing its memory footprint. This way, RMLStreamer allows efficiently generating Linked Data in streaming use cases as well.

In the use case demonstrator, RMLStreamer is evaluated on a homecare monitoring use case with multiple sensors generating JSON data observations every second. The results prove that RMLStreamer can very efficiently process these JSON observations and map them to RDF data. For a scenario where 14 sensor observations per second are generated, the average processing time is only 13.5 ms.

To summarize, hypothesis (a) of this chapter can be validated by following the Linked Data approach and using technologies and tools such as RML and RMLStreamer.

6.4.2 Service providers

Service providers are responsible to build services upon the data exposed by the data providers. In existing non-semantic systems, custom non-reusable services are often built. This leads to static systems that require much manual configuration effort. Different semantic building blocks such as Streaming MASSIF and DIVIDE in combination with engines such as C-Sprite or C-SPARQL allow moving away from this.

Both Streaming MASSIF and DIVIDE take the available background knowledge and contextual data of the patient profiles into account when performing semantic reasoning. This way, these tools allow designing personalized services. Moreover, they are both designed to deal with the huge amount of high-volume and high-velocity data that is coming in on the data streams in many healthcare monitoring use cases. They are designed for a distributed cascading reasoning architecture, where the processing of the raw data streams is not done on one big, centralized, monolithic server. Instead, according to the use case requirements, some data stream processing might already be performed in the edge of the IoT network, for example on a device in the local environment of the patient. This is done in the selection layer of Streaming MASSIF, where different engines such as C-Sprite, C-SPARQL or another regular RSP engine can be employed. C-Sprite is especially useful when efficient reasoning needs to be performed with many hierarchical concepts.

DIVIDE is responsible for configuring the queries that are evaluated on a (local) C-Sprite or C-SPARQL engine. Through the defined generic DIVIDE query templates, DIVIDE deploys those specific RSP queries that are relevant with the given environmental use case context. To do this, it performs semantic reasoning on the domain knowledge and relevant context information such as the patient's profile, every time this context changes. The evaluation results on the use case demonstrator show that the query derivation for that use case takes a little over 7 seconds. This is relatively high, but it is important to realize that this query derivation is only performed upon context changes such as medical profile updates. The frequency of such

changes is a few orders of magnitudes smaller than the frequency of the RSP query evaluation. DIVIDE ensures that only the relevant data is filtered, and that no real-time reasoning is required during the query evaluation. Hence, this query evaluation is very efficient. This is shown in the C-SPARQL evaluation results, which report an average query execution time of only 26.4 ms for a data stream containing 14 RDF observations per second. It should be noted that the query evaluation is also performant on low-end devices with few resources, even for data streams with a much higher data velocity [28]. This is especially important in the distributed context in which is DIVIDE is designed to be employed, since edge processing devices in IoT often have only a limited number of resources. Hence, DIVIDE allows the local execution of queries in a challenging IoT environment. This allows for improved system performance, scalability, local autonomy and enables data privacy by design [61].

In its abstraction and temporal reasoning layer, Streaming MASSIF allows easily defining functionality through new semantic axioms and rules. As shown in the implementation details of the use case demonstrator, definitions for certain event types such as a high fever event and a rising fever event can be defined in a relatively simple way. A semantic reasoner is then used to derive new knowledge through these definitions out of the data coming in from the selection layer. Similarly for the services instructed on top of these layers, simple queries can be defined to describe the functionality of the service. By semantically defining these definitions and queries, the functionality of services is semantically clear. This allows them to be reused in a user-friendly way. Streaming MASSIF also delivers performant semantic services, as is shown through the evaluation results on the use case demonstrator. On average, it takes a little over 1.5 seconds to generate the correct notification corresponding to a fever event received from the selection layer. Considering this processing includes expressive semantic reasoning on the full ontology with all medical domain knowledge and Rosa's profile information, this is a performant result.

Based on this discussion, it can be concluded that hypothesis (b) of this chapter can be validated by using the DIVIDE and Streaming MASSIF building blocks in a distributed, cascading reasoning architecture. More specifically, Streaming MASSIF validates sub-hypothesis (i), while using DIVIDE allows validating sub-hypothesis (ii).

6.4.3 Integrators

Integrators are used to compose workflows that fulfill a particular functionality. Existing non-semantic systems typically allow building generic, static workflows. In addition, they are often still constructed manually. This makes it cumbersome and nearly infeasible to coordinate these workflows across different organizations and stakeholders involved in the follow-up and medical treatment of patients. AMADEUS solves these issues by using semantics. More specifically, it requires that all context and

profile information, possible workflow steps and policies, and any other relevant information to construct possible workflows is semantically described. It allows semantically defining the functionality of services and policies, and the quality parameters offered by them. This information is taken into account when possible workflows are being composed through semantic reasoning. This way, the resulting workflow is guaranteed to offer the desired functionality and to meet the end user's quality constraints, which can be dynamically chosen.

To make this more tangible, the use case demonstrator focuses on one particular example where workflows represent medical treatment plans for a disease. In this example, possible workflow steps and policies are represented by potential steps in the treatment of different diseases. For each such step, it can be semantically defined when applying this step is useful (e.g., for which diseases, given which preconditions), what the impact on the state and context is (e.g., how much does it cure the patient's disease or influence the actual diagnosis), what the quality parameters of this step are (e.g., what is the patient comfort, the treatment cost, or the impact on the relapse risk after a few years), and what possible contraindications exist for this step (e.g., what medication cannot be used or what other diagnoses cannot be present to take this step). This makes it possible to create personalized, dynamic treatment plans (workflows) that take into account particular quality constraints about the treatment.

An additional advantage of using AMADEUS is its ability to perform automatic conflict detection between workflows. In the use case demonstrator example, conflicts can exist if the current profile or treatment plan of a patient represents a contraindication for another treatment plan. In the demonstrator scenario, this was the case when Rosa got an influenza infection, yielding a conflict with her existing treatment plan. This plan was in place to treat here colon cancer, and contained a chemoradiotherapy session scheduled before her influenza could be cured. The automatic detection of such conflicts is particularly interesting in cross-organizational environments, like in the use case scenario. The original colon cancer treatment plan was constructed by a hospital doctor, while the new influenza treatment plan was created by Rosa's GP. Hence, this demonstrates how AMADEUS can help improving the communication and coordination of workflows across the different organizations and stakeholders involved in Rosa's caregiving.

The evaluation results of AMADEUS on the use case demonstrator show that AMADEUS can efficiently generate its dynamic workflows. All possible treatment plans for both the colon cancer and influenza diagnoses are generated in less than 200 ms on average. The conflict detection takes on average a little above 1.3 s, which is still very acceptable given the fact that AMADEUS should not be deployed in a real-time data processing pipeline.

To summarize, it can be concluded that the design and performance of AMADEUS allows validating hypothesis (c) of this chapter.

6.4.4 Installers

Installers are the people responsible for configuring all data provisioning tools, services, and workflows in the continuous homecare provisioning system. For the first two aspects, GUIs built on top of the existing semantic tools are available.

To generate Linked Data from various heterogeneous data sources, RML mapping rules can be used. Defining such rules is however a tedious and time-consuming work, as mappings need to be created for each type of input data source to the designed semantic ontology model. To make this process much easier, either the RMLEditor or Matey can be used. These tools have an optimized GUI to easily generate mapping rules, visualize the resulting Linked Data on selected input data sources, and export the corresponding RML mapping rules to be used by the actual Linked Data generation tools. Matey is most suited for developers who do not have knowledge about Semantic Web technologies, while the RMLEditor is most useful for data owners who are no developers.

To configure the axioms, rules and queries that define the services in Streaming MASSIF, a GUI is also available. This GUI allows installers to easily enter these definitions, without having to bother with the technological details of the underlying system. However, additional research is still needed to design a UI to properly configure DIVIDE and its generic DIVIDE queries.

To conclude, the available UIs for Streaming MASSIF and the RML mapping rule generation validate hypothesis (d) of this chapter.

6.5 Conclusion

The impact and contribution of this chapter is that it brings together different existing building blocks, built upon Semantic Web technologies, into a reference architecture that can be leveraged to optimize continuous homecare provisioning use cases. To this end, a distributed, cascading reasoning architecture is designed. This architecture allows solving the challenges associated to the different roles involved in continuous care solutions. For data providers, the architecture allows exposing data as Linked Data to services and other organizations in reusable fashion, using declarative mapping rules. This Linked Data can be efficiently generated in use cases dealing with high-velocity streaming data. Concerning service providers, the architecture allows designing dynamic, use case specific, data-driven, personalized, reusable services. These services are defined by declaratively expressing their functionality and meaning as semantic definitions, and operate on the data abstractions and insights generated by stream reasoning queries. These queries efficiently process the generated Linked Data in a cascading reasoning pipeline, which allows for improved performance, scalability, local autonomy and data privacy of the system. Moreover, considering service integrators, the architecture allows constructing dynamic workflows of different services

or specific functionality described through declarative semantic descriptions. Conflicts can be automatically detected between constructed workflows, improving their coordination across organizations and stakeholders involved in the care provisioning of patients. By chaining all building blocks, a feedback loop is created: knowledge generated through services and workflows can result in context changes, which are automatically reflected in the adaptive, context-aware stream reasoning queries. Finally, for installers, different UIs are available to easily expose Linked Data and build dynamic services in a user-friendly way. This allows installers to configure the system without requiring knowledge about technical details, minimizing the manual effort and risk of configuration errors. Through the performance evaluation on a use case demonstrator, the chapter has also shown that the different building blocks of the reference architecture can perform their tasks in an efficient way.

Future work could include the application and validation of the presented reference architecture on other healthcare use cases, as well as investigating its generalization towards other applications domains. Moreover, more research is required to extend the available tools and UIs for installers. An important example of this is investigating how a UI can be designed for DIVIDE, to properly configure DIVIDE and its different generic queries.

Funding

This research is partly funded by FWO (Research Foundation – Flanders) SBO grant 150038 (DiSSeCt). This work is also partly funded by the postdoctoral fellowship of FWO of Pieter Bonte (1266521N) and Ruben Verborgh (12I9219N).

Availability of data and materials

Code and set-up documentation of the different tools are available online. For DIVIDE, this information is available at https://github.com/IBCNServices/DIVIDE. For Streaming MASSIF, this is provided at https://github.com/IBCNServices/StreamingMASSIF. For AMADEUS, this is available at https://github.com/IDLabResearch/AMADEUS-workflows. For the RMLStreamer and RMLMapper, more information is available at https://github.com/RMLio/RMLStreamer and https://github.com/RMLio/rmlmapper-java, respectively. Similarly, https://rml.io/tools/rmleditor/ and https://w3id.org/yarrrml/matey/ contain extra information about the RMLEditor and Matey, respectively. For C-Sprite, extra info is available at https://github.com/IBCNServices/C-Sprite. The ACCIO ontology used in the use case demonstrator is available online at https://github.com/IBCNServices/Accio-Ontology/tree/gh-pages.

References

[1] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. *Context Aware Computing for The Internet of Things: A Survey*. IEEE Communications Surveys & Tutorials, 16(1):414–454, 2014. doi:10.1109/SURV.2013.042313.00197.

[2] O. B. Sezer, E. Dogdu, and A. M. Ozbayoglu. *Context-Aware Computing, Learning, and Big Data in Internet of Things: A Survey*. IEEE Internet of Things Journal, 5(1):1–27, 2018. doi:10.1109/JIOT.2017.2773600.

[3] K. Avila, P. Sanmartin, D. Jabba, and M. Jimeno. *Applications based on service-oriented architecture (SOA) in the field of home healthcare*. Sensors, 17(8), 2017. doi:10.3390/s17081703.

[4] J. Emanuele and L. Koetter. *Workflow opportunities and challenges in healthcare*. 2007 BPM & Workflow Handbook, 2007.

[5] T. Zayas-Cabán, S. N. Haque, and N. Kemper. *Identifying Opportunities for Workflow Automation in Health Care: Lessons Learned from Other Industries*. Applied Clinical Informatics, 12(03):686–697, 2021. doi:10.1055/s-0041-1731744.

[6] K. Van den Bosch, P. Willemé, J. Geerts, J. Breda, S. Peeters, S. Van De Sande, F. Vrijens, C. Van de Voorde, and S. Stordeur. *Residential care for older persons in Belgium: Projections 2011–2025 – Supplement*. KCE Reports 167C, Belgian Health Care Knowledge Centre (KCE), 2011. Available from: https://kce.fgov.be/sites/default/files/atoms/files/KCE_167S_residential_elderly_care_supplement.log_.pdf.

[7] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. *Semantics for the Internet of Things: Early Progress and Back to the Future*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012. doi:10.4018/jswis.2012010101.

[8] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride. *RDF 1.1 concepts and abstract syntax*. W3C Recommendation, World Wide Web Consortium (W3C), 2014. Available from: https://www.w3.org/TR/rdf11-concepts/.

[9] W3C OWL Working Group. *OWL 2 Web Ontology Language*. W3C Recommendation, World Wide Web Consortium (W3C), 2012. Available from: https://www.w3.org/TR/owl2-overview/.

[10] T. R. Gruber. *A translation approach to portable ontology specifications*. Knowledge Acquisition, 5(2):199–220, 1993. doi:10.1006/knac.1993.1008.

[11] C. Bizer, T. Heath, and T. Berners-Lee. *Linked data: The story so far*. In A. Sheth, editor, Semantic Services, Interoperability and Web Applications: Emerging

Concepts, pages 205–227. IGI Global, 2011. doi:10.4018/978-1-60960-593-3.ch008.

[12] S. Harris and A. Seaborne. *SPARQL 1.1 Query Language*. W3C Recommendation, World Wide Web Consortium (W3C), 2013. Available from: https://www.w3.org/TR/sparql11-query/.

[13] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. *RDFox: A highly-scalable RDF store*. In The Semantic Web - ISWC 2015: Proceedings, Part II of the 14th International Semantic Web Conference, pages 3–20, Cham, Switzerland, 2015. Springer. doi:10.1007/978-3-319-25010-6_1.

[14] J. Urbani, C. Jacobs, and M. Krötzsch. *Column-oriented datalog materialization for large knowledge graphs*. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. OJS/PKP, 2016. doi:10.1609/aaai.v30i1.9993.

[15] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. *Stream reasoning: A survey and outlook*. Data Science, 1(1-2):59–83, 2017. doi:10.3233/DS-170006.

[16] X. Su, E. Gilman, P. Wetz, J. Riekki, Y. Zuo, and T. Leppänen. *Stream reasoning for the Internet of Things: Challenges and gap analysis*. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), pages 1–10, New York, NY, USA, 2016. Association for Computing Machinery (ACM). doi:10.1145/2912845.2912853.

[17] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte, and O. Corcho. *RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems*. International Journal on Semantic Web and Information Systems (IJSWIS), 10(4):17–44, 2014. Available from: https://dl.acm.org/doi/10.5555/2795081.2795083.

[18] D. Van Assche, T. Delva, G. Haesendonck, P. Heyvaert, B. De Meester, and A. Dimou. *Declarative RDF graph generation from heterogeneous (semi-)structured data: A systematic literature review*. Journal of Web Semantics, 75, 2023. doi:10.1016/j.websem.2022.100753.

[19] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle. *RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data*. In Proceedings of the Workshop on Linked Data on the Web, co-located with the 23rd International World Wide Web Conference (WWW 2014), volume 1184, 2014. Available from: https://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[20] F. Michel, L. Djimenou, C. Faron-Zucker, and J. Montagnat. *Translation of Relational and Non-relational Databases into RDF with xR2RML*. In Proceedings of the

11th International Conference on Web Information Systems and Technologies - WEBIST, pages 443–454, 2015. doi:10.5220/0005448304430454.

[21] A. Chortaras and G. Stamou. *Mapping Diverse Data to RDF in Practice*. In The Semantic Web – ISWC 2018, pages 441–457. Springer International Publishing, 2018. doi:10.1007/978-3-030-00671-6_26.

[22] B. Vu, J. Pujara, and C. A. Knoblock. *D-REPR: A Language for Describing and Mapping Diversely-Structured Data Sources to RDF*. In Proceedings of the 10th International Conference on Knowledge Capture, pages 189–196, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3360901.3364449.

[23] S. Das, S. Sundara, and R. Cyganiak. *R2RML: RDB to RDF Mapping Language*. W3C Recommendation, World Wide Web Consortium (W3C), 2012. Available from: https://www.w3.org/TR/rdf11-concepts/.

[24] A. Dimou, T. De Nies, R. Verborgh, E. Mannens, and R. Van de Walle. *Automated Metadata Generation for Linked Data Generation and Publishing Workflows*. In S. Auer, T. Berners-Lee, C. Bizer, and T. Heath, editors, Proceedings of the 9th Workshop on Linked Data on the Web, CEUR Workshop Proceedings, 2016.

[25] S. Jozashoori and M.-E. Vidal. *MapSDI: A Scaled-Up Semantic Data Integration Framework for Knowledge Graph Creation*. In On the Move to Meaningful Internet Systems: OTM 2019 Conferences, pages 58–75. Springer International Publishing, 2019. doi:10.1007/978-3-030-33246-4_4.

[26] K. Kyzirakos, D. Savva, I. Vlachopoulos, A. Vasileiou, N. Karalis, M. Koubarakis, and S. Manegold. *GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings*. Journal of Web Semantics, 52-53:16–32, 2018. doi:10.1016/j.websem.2018.08.003.

[27] G. Haesendonck, W. Maroy, P. Heyvaert, R. Verborgh, and A. Dimou. *Parallel RDF generation from heterogeneous big data*. In SBD '19: Proceedings of the International Workshop on Semantic Big Data, pages 1–6, 2019. doi:10.1145/3323878.3325802.

[28] M. De Brouwer, B. Steenwinckel, Z. Fang, M. Stojchevska, P. Bonte, F. De Turck, S. Van Hoecke, and F. Ongenae. *Context-aware query derivation for IoT data streams with DIVIDE enabling privacy by design*. Semantic Web, 14(5):893–941, 2023. doi:10.3233/SW-223281.

[29] M. De Brouwer, D. Arndt, P. Bonte, F. De Turck, and F. Ongenae. *DIVIDE: Adaptive Context-Aware Query Derivation for IoT Data Streams*. In Joint Proceedings of the International Workshops on Sensors and Actuators on the Web, and

Semantic Statistics, co-located with the 18th International Semantic Web Conference (ISWC 2019), volume 2549, pages 1–16, Aachen, 2019. CEUR Workshop Proceedings. Available from: https://ceur-ws.org/Vol-2549/article-01.pdf.

[30] K. Jaiswal and V. Anand. *A Survey on IoT-Based Healthcare System: Potential Applications, Issues, and Challenges*. In A. A. Rizvanov, B. K. Singh, and P. Ganasala, editors, Advances in Biomedical Engineering and Technology, pages 459–471. Springer Singapore, 2021. doi:10.1007/978-981-15-6329-4_38.

[31] R. Zgheib, S. Kristiansen, E. Conchon, T. Plageman, V. Goebel, and R. Bastide. *A scalable semantic framework for IoT healthcare applications*. Journal of Ambient Intelligence and Humanized Computing, 2020. doi:10.1007/s12652-020-02136-2.

[32] S. Jabbar, F. Ullah, S. Khalid, M. Khan, and K. Han. *Semantic interoperability in heterogeneous IoT infrastructure for healthcare*. Wireless Communications and Mobile Computing, 2017, 2017. doi:10.1155/2017/9731806.

[33] F. Ullah, M. A. Habib, M. Farhan, S. Khalid, M. Y. Durrani, and S. Jabbar. *Semantic interoperability for big-data in heterogeneous IoT infrastructure for healthcare*. Sustainable Cities and Society, 34:90–96, 2017. doi:10.1016/j.scs.2017.06.010.

[34] V. Subramaniyaswamy, G. Manogaran, R. Logesh, V. Vijayakumar, N. Chilamkurti, D. Malathi, and N. Senthilselvan. *An ontology-driven personalized food recommendation in IoT-based healthcare system*. The Journal of Supercomputing, 75(6):3184–3216, 2019. doi:10.1007/s11227-018-2331-8.

[35] P. Bonte, R. Tommasini, E. Della Valle, F. De Turck, and F. Ongenae. *Streaming MASSIF: cascading reasoning for efficient processing of iot data streams*. Sensors, 18(11):3832, 2018. doi:10.3390/s18113832.

[36] H. Stuckenschmidt, S. Ceri, E. Della Valle, and F. Van Harmelen. *Towards expressive stream reasoning*. In Semantic Challenges in Sensor Networks, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. doi:10.4230/DagSemProc.10042.4.

[37] P. Bonte, R. Tommasini, F. De Turck, F. Ongenae, and E. D. Valle. *C-Sprite: Efficient Hierarchical Reasoning for Rapid RDF Stream Processing*. In Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, pages 103–114, 2019. doi:10.1145/3328905.3329502.

[38] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. *C-SPARQL: a continuous query language for RDF data streams*. International Journal of Semantic Computing, 4(1):3–25, 2010. doi:10.1142/S1793351X10000936.

[39] J.-P. Calbimonte, O. Corcho, and A. J. Gray. *Enabling ontology-based access to streaming data sources*. In The Semantic Web – ISWC 2010, pages 96–111. Springer, 2010. doi:10.1007/978-3-642-17746-0_7.

[40] R. Tommasini and E. Della Valle. *Yasper 1.0: Towards an RSP-QL Engine*. In Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks, co-located with 16th International Semantic Web Conference (ISWC 2017). CEUR Workshop Proceedings, 2017. Available from: https://ceur-ws.org/Vol-1963/paper487.pdf.

[41] R. Tommasini, P. Bonte, F. Ongenae, and E. Della Valle. *RSP4J: An API for RDF Stream Processing*. In R. Verborgh, K. Hose, H. Paulheim, P.-A. Champin, M. Maleshkova, O. Corcho, P. Ristoski, and M. Alam, editors, The Semantic Web: Proceedings of the 18th International Conference, ESWC 2021, pages 565–581, Cham, Switzerland, 2021. Springer. doi:10.1007/978-3-030-77385-4_34.

[42] H. Sun, D. Arndt, J. De Roo, and E. Mannens. *Predicting future state for adaptive clinical pathway management*. Journal of Biomedical Informatics, 117, 2021. doi:10.1016/j.jbi.2021.103750.

[43] *AMADEUS*, 2020. Accessed: 2020-04-01. Available from: https://github.com/IDLabResearch/AMADEUS-workflows.

[44] H. Sun, K. Depraetere, J. De Roo, G. Mels, B. De Vloed, M. Twagirumukiza, and D. Colaert. *Semantic processing of EHR data for clinical research*. Journal of Biomedical Informatics, 58:247–259, 2015. doi:10.1016/j.jbi.2015.10.009.

[45] Y.-F. Zhang, Y. Tian, T.-S. Zhou, K. Araki, and J.-S. Li. *Integrating HL7 RIM and ontology for unified knowledge and data representation in clinical decision support systems*. Computer Methods and Programs in Biomedicine, 123:94–108, 2016. doi:10.1016/j.cmpb.2015.09.020.

[46] D. A. Alexandrou, I. E. Skitsas, and G. N. Mentzas. *A Holistic Environment for the Design and Execution of Self-Adaptive Clinical Pathways*. IEEE Transactions on Information Technology in Biomedicine, 15(1):108–118, 2011. doi:10.1109/TITB.2010.2074205.

[47] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler. *N3Logic: A logical framework for the World Wide Web*. Theory and Practice of Logic Programming, 8(3):249–269, 2008. doi:10.1017/S1471068407003213.

[48] R. Verborgh and J. De Roo. *Drawing Conclusions from Linked Data on the Web: The EYE Reasoner*. IEEE Software, 32(3):23–27, 2015. doi:10.1109/MS.2015.63.

[49] P. Bonte, F. Ongenae, J. Nelis, T. Vanhove, and F. De Turck. *User-Friendly and Scalable Platform for the Design of Intelligent IoT Services: a Smart Office Use Case*. In Proceedings of the ISWC 2016 Posters & Demonstrations Track, co-located with 15th International Semantic Web Conference (ISWC 2016), 2016. Available from: https://ceur-ws.org/Vol-1690/paper99.pdf.

[50] P. Heyvaert, A. Dimou, A.-L. Herregodts, R. Verborgh, D. Schuurman, E. Mannens, and R. Van de Walle. *RMLEditor: A Graph-Based Mapping Editor for Linked Data Mappings*. In The Semantic Web: Latest Advances and New Domains: Proceedings of the 13th International Conference, ESWC 2016, pages 709–723. Springer International Publishing, 2016. doi:10.1007/978-3-319-34129-3_43.

[51] P. Heyvaert, A. Dimou, B. D. Meester, T. Seymoens, A.-L. Herregodts, R. Verborgh, D. Schuurman, and E. Mannens. *Specification and implementation of mapping rule visualization and editing: MapVOWL and the RMLEditor*. Journal of Web Semantics, 49:31–50, 2018. doi:10.1016/j.websem.2017.12.003.

[52] P.-Y. Vandenbussche, G. A. Atemezing, M. Poveda-Villalón, and B. Vatant. *Linked Open Vocabularies (LOV): A gateway to reusable semantic vocabularies on the Web*. Semantic Web, 8(3):437–452, 2017. doi:10.3233/SW-160213.

[53] P. Heyvaert, B. De Meester, A. Dimou, and R. Verborgh. *Declarative Rules for Linked Data Generation at Your Fingertips!* In The Semantic Web: ESWC 2018 Satellite Events, pages 213–217. Springer, 2018. doi:10.1007/978-3-319-98192-5_40.

[54] imec – Ghent University – IDLab. *YARRRML*, 2017. Accessed: 2023-04-24. Available from: https://rml.io/yarrrml/.

[55] O. Ben-Kiki, C. Evans, and B. Ingerson. *Yaml Ain't Markup Language (YAML), version 1.2*. Technical report, 2009. Available from: https://yaml.org/spec/1.2.2/.

[56] M. De Brouwer, P. Bonte, D. Arndt, M. Vander Sande, P. Heyvaert, A. Dimou, R. Verborgh, F. De Turck, and F. Ongenae. *Distributed Continuous Home Care Provisioning through Personalized Monitoring & Treatment Planning*. In Companion Proceedings of the Web Conference 2020 (WWW 2020), pages 143–147. Association for Computing Machinery (ACM), 2020. doi:10.1145/3366424.3383528.

[57] A. Taha, I. Vinograd, A. Sakhnini, N. Eliakim-Raz, L. Farbman, R. Baslo, S. M. Stemmer, A. Gafter-Gvili, L. Leibovici, and M. Paul. *The association between infections and chemotherapy interruptions among cancer patients: Prospective cohort study*. Journal of Infection, 70(3):223–229, 2015. doi:10.1016/j.jinf.2014.10.008.

[58] F. Ongenae, P. Duysburgh, N. Sulmon, M. Verstraete, L. Bleumers, S. De Zutter, S. Verstichel, A. Ackaert, A. Jacobs, and F. De Turck. *An ontology co-design method for the co-creation of a continuous care ontology*. Applied Ontology, 9(1):27–64, 2014. doi:10.3233/AO-140131.

[59] M. De Brouwer, F. Ongenae, P. Bonte, and F. De Turck. *Towards a Cascading Reasoning Framework to Support Responsive Ambient-Intelligent Healthcare Interventions*. Sensors, 18(10):3514, 2018. doi:10.3390/s18103514.

[60] K. Donnelly. *SNOMED-CT: The advanced terminology and coding system for eHealth*. Studies in Health Technology and Informatics, 121, 2006.

[61] A. Cavoukian. *Privacy by design*, 2009. Accessed: 2022-09-25. Available from: https://www.ipc.on.ca/wp-content/uploads/Resources/7foundationalprinciples.pdf.

7

Conclusions

This dissertation has investigated how adaptive and performant semantic reasoning can be performed on Internet of Things (IoT) data streams in IoT applications, with a focus on the healthcare application domain, to solve the shortcomings and issues associated with this task in the current state-of-the-art. To this end, the different chapters have investigated the research challenges presented in Section 1.5, by zooming in on the contributions and validating the research hypotheses discussed in Section 1.6. The healthcare application domain was chosen to evaluate the contributions of this dissertation and validate the research hypotheses, by considering the different healthcare use cases introduced in Section 1.6.

In summary, Chapter 2 has presented a generic cascading reasoning framework, enabling semantic stream reasoning for IoT applications in a responsive manner, with the introduction of local autonomy. To make the conditions and window parameters of stream processing queries in such a cascading reasoning architecture adaptive to changing use case context, Chapter 3 has introduced the semantic IoT platform component DIVIDE. DIVIDE also allows end users to integrate privacy by design into applications built with the cascading reasoning framework. Chapter 4 has applied the cascading reasoning framework with DIVIDE to an additional use case to demonstrate its generic design. In Chapter 5, the design of DIVIDE has been further extended to also make the stream processing queries adaptive to constantly varying situational context, such as network characteristics and query performance, in a per use case configurable way. This way, DIVIDE can be adaptive to the full environmental context in which the stream processing queries of a semantic IoT platform

are being evaluated. Finally, Chapter 6 has demonstrated how DIVIDE can be embedded as a building block into a full semantic platform together with other semantic components across a cascading reasoning architecture. It has zoomed in on such a platform for the healthcare domain, to show how it can optimize continuous care and help close the loop in IoT application domains like healthcare.

To conclude this doctoral dissertation, this chapter summarizes the previous chapters, and reflects on them in relation to the dissertation's research challenges, contributions and hypotheses. Moreover, it identifies open challenges and possible future directions in the addressed research fields.

7.1 Review of the research challenges, contributions and hypotheses

In Section 1.5, the problem statement of this doctoral dissertation was summarized into four research challenges. In Section 1.6, four research contributions were listed as a solution to the presented challenges. For every contribution, one or multiple research hypotheses were defined. This section reflects on the different challenges and discusses how the contributions and their evaluation on the chosen healthcare use cases, as presented in the previous chapters, allow validating the research hypotheses.

Research challenge RCH1: Performant & responsive real-time stream reasoning with local autonomy across a heterogeneous IoT network

To address the first research challenge of this dissertation, a generic cascading reasoning framework is designed in Chapter 2. This represents research contribution RCO1 of this dissertation. The presented cascading reasoning framework moves away from centralized processing architectures by realizing the vision of cascading reasoning in a responsive and easily applicable manner, while also allowing for local autonomy. The framework allows easily constructing an application-specific network of stream reasoning components that can be hosted locally, in the edge of the network, and in a central server or cloud environment. It allows using heterogeneous processing devices in the constructed pipeline, e.g., a low-end device with a limited number of resources as the local processing device. This way, the framework aligns with the principles of edge & fog computing.

Using the cascading reasoning framework in a stream reasoning system has multiple advantages. First of all, the framework addresses the performance trade-off in stream reasoning between reasoning expressivity and data velocity. It allows solving this challenge by letting the first components in the network of stream reasoning components mainly perform filtering with no or low expressivity reasoning, while the subsequent components increase the expressivity of the semantic reasoning as the data velocity decreases. Moreover, the framework allows introducing local autonomy,

since the queries on local and edge devices of the network can also derive (actionable) insights and can decide whether or not any relevant events or insights should be forwarded to the central devices. Furthermore, additional advantages of the framework include that it results in minimal network congestion by reducing the traffic as less data is transferred due to local processing, it saves central resources of the network for optimal usage in high priority situations as some processing is delegated to the local & edge components, and it removes a single point of failure as the processing is no longer done by a single central component.

As is shown in Chapter 2, the architecture of the generic cascading reasoning framework fits within a generic reference architecture for Ambient Assisted Living (AAL) and Enhanced Living Environments (ELE) platforms [1]. It is the first framework in the domains of AAL and ELE that combines the principles of semantic stream reasoning, cascading reasoning and edge computing. This way, the framework also allows dealing with multiple shortcomings associated with existing centralized processing architectures.

In Chapter 2, the cascading reasoning framework is applied to the pervasive healthcare use case UC1 about the hospital monitoring of patients. A performance evaluation of the framework has shown that the framework can detect alarming situations according to the diagnosis of the monitored patient, based on a combination of real-time sensor data, context information and medical domain knowledge. This evaluation uses heterogeneous processing devices in a constructed pipeline, including a low-end device with a limited number of resources as the local processing device. The results show that whenever the detected alarming situation requires a nurse to be called according to the definitions in the ontologies, the nurse call (i.e., the (result from the) actionable insights) can be generated by the deployed pipeline of stream reasoning components in on average 3.1 seconds after the alarming situation first begins. These results are obtained when performing rule-based reasoning on the edge & central reasoning components of the set-up using the complex ACCIO continuous care ontology [2]. Replacing the simplified nurse selection algorithm of the evaluation queries on the central reasoning component with a realistic, more complex algorithm would require an additional 0.55 seconds to be added to this number [3]. This implies that the evaluated set-up can complete a nurse call assignment in less than 5 seconds, which ensures that the system meets the demands in various countries that such alarms should be handled by a nurse within 5 minutes after the alarming situation starts. In general, this proves that the cascading reasoning framework can be applied to healthcare and AAL use cases that require responsive, performant real-time processing of IoT sensor data streams.

Following the observations about the framework and the presented evaluation results, it is clear that **research contribution RCO1 allows validating research hypothesis RH1**: "The realization of a generic cascading reasoning framework in an IoT network will improve the overall performance of semantic stream reasoning on

IoT data streams. The full pipeline of stream reasoning components will be able to generate relevant actionable insights from events in the data and handle those events in less than 5 seconds.".

The evaluations in Chapter 2 also demonstrate that the cascading reasoning framework adds local autonomy to the system: if the current context implies that no nurse should be called for an alarming situation, the system only generates a warning to the nurse and derives actionable insights about what changes should happen to the environment, e.g., dimming the lights in the room. The results show that these insights can be generated in on average 2.7 seconds, which is also below the threshold of 5 seconds to handle an alarming situation. The derivation of these insights is performed by the stream reasoning engines on the local & edge devices of the set-up, and does not involve the central stream reasoning engine.

Considering the presented evaluation results, it is also clear that **research contribution RCO1 allows validating research hypothesis RH2**: "The realization of a generic cascading reasoning framework in an IoT network will introduce local autonomy by letting local & edge devices in the network host queries. This will allow certain events in the data to be handled locally through actionable insights derived from the data, without requiring human intervention or involving central reasoning components. The local & edge components in the pipeline will also be able to perform these tasks in less than 5 seconds.".

Appendix A focuses on the local components of the cascading reasoning framework for use case UC4 about the continuous monitoring of amateur cyclists. The designed real-time feedback system gives personalized feedback about a rider's heart rate and heart rate zones according to the rider's profile. A performance evaluation of the designed system on a low-end device with limited resources shows that continuous real-time feedback can be given on a single low-end device every 5 seconds for 12 cyclists at most. This implies that the execution times of the feedback queries remain below 5 seconds for this number of riders. When only considering up to 5 riders, continuous feedback is even possible every second.

The generic design of the cascading reasoning framework allows extrapolating the presented findings to employ research contribution RCO1 for other IoT application domains. Generally speaking, the framework can be applied to any IoT application that needs to deal with streaming data. For every individual use case, the optimal network of stream reasoning components should be manually constructed. It should be noted here that not every component type in the architecture of the framework (i.e., RSPS, LRS or BRS) should necessarily be included in this network, while it is also possible to chain multiple components of the same type. This is possible due to the modular design of the components and the overall modifiability of the framework at design time. By matching the number of components in the pipeline to the size of the use case, optimal scalability and performance can be achieved. On every component, this performance is impacted by the expressivity of the ontology, the velocity

of the data, the complexity of the queries, and criteria specific to the device (i.e., resource specifications) and query evaluation & reasoning engine (i.e., implementation details). Moreover, the impact of the network latency across the full IoT network also influences the overall performance of the system. Hence, all these aspects need to be considered when constructing the optimal cascading network. In general, a larger network of stream reasoning components leads to more fine-grained control over the data flow and query distribution, at the expense of increased complexity in the management of those queries and the semantic data across the system. Furthermore, local autonomy can be easily introduced into every application through the design of the queries on the local and edge devices of the network: those queries define which (actionable) insights can be derived on these devices, and in which cases certain insights or events are further propagated to the central components.

Research challenge RCH2: Adaptive configuration of stream processing queries based on use case context, enabling privacy by design

To address the second research challenge of this dissertation, the semantic IoT platform component DIVIDE is designed. In Chapter 3, the full methodological design of DIVIDE is discussed. This represents research contribution RCO2 of this dissertation. DIVIDE should be used in a semantic IoT platform with an architecture that applies the cascading reasoning framework presented in Chapter 2. DIVIDE can automatically and adaptively derive and manage the queries of the stream processing components in the platform. It is adaptive and context-aware by design: it performs semantic reasoning to derive the contextually relevant queries for every individual component whenever changes are observed to the use case context that is relevant to that component. This reasoning is performed using generic DIVIDE queries, which specify how the conditions and window parameters should be instantiated to actual stream processing queries according to the current use case context. These generic DIVIDE query definitions can be easily configured by end users from existing generic queries that are typically evaluated on centralized processing architectures. This is the only query configuration that DIVIDE requires from its end users: due to its adaptiveness, no manual reconfiguration of stream processing queries is required whenever the use case context changes. Importantly, the methodological design of DIVIDE results in simple stream processing queries that do not require any more semantic reasoning during their evaluation, and can thus be efficiently evaluated. The DIVIDE methodology has also been realized in a first implementation.

The methodological design of DIVIDE also enables privacy by design. It helps end users to integrate privacy by design into IoT applications that deal with streaming data and employ DIVIDE in a semantic IoT platform. DIVIDE enables this by leaving its end users in full control to specifically define which data, both raw data and data abstractions in the outputs of the semantic queries, can leave the local environments of the network. The end user can control this by exactly defining via the

generic DIVIDE query templates which semantic concepts are filtered by the local stream processing engines and will thus be sent over the network. Only the outputs of those queries will be sent over the network, and all other data will be kept locally.

In Chapter 3, a first implementation of DIVIDE is evaluated on the homecare monitoring use case UC2. In this evaluation, the in-home routine and non-routine activities of patients are monitored based on the patient's in-home location. The evaluation compares the query execution times of stream processing queries derived by DIVIDE with the same measures for state-of-the-art stream reasoning set-ups. The results demonstrate that DIVIDE performs comparable or even slightly better in terms of query performance. On a regular processing device, for a query that is detecting toileting activities, the evaluation of the queries derived by DIVIDE on a C-SPARQL RDF Stream Processing (RSP) engine [4] takes on average 0.295 seconds over the full evaluation, versus 0.716 seconds for the real-time reasoning queries on a streaming version of the state-of-the-art RDFox semantic reasoning engine [5]. Similarly, for a query detecting brushing teeth activities, these numbers are 0.226 seconds and 0.423 seconds, respectively. Additional evaluation results prove that the queries derived by DIVIDE can be evaluated on a C-SPARQL engine that is running on a low-end device with limited resources in on average 3.666 seconds and 3.001 seconds for the toileting and brushing teeth activities, respectively.

From the methodological design of DIVIDE and the presented evaluation results, it can be concluded that **research contribution RCO2 allows validating research hypothesis RH3**: "The methodological design of a semantic IoT platform component that derives and configures the conditions & window parameters of stream processing queries whenever the use case context changes will result in adaptive, context-aware queries that only require simple filtering and thus enable the local filtering of contextually relevant events in less than 5 seconds on low-end IoT devices with few resources. This will fully remove the required manual query reconfiguration effort when changes to the use case context occur.".

The evaluation results on the homecare monitoring use case UC2 presented in Chapter 3 also report the overhead of deriving queries with DIVIDE when the use case context changes. In the given use case, the frequency of such context changes, and thus the execution frequency of the query derivation, is at least an order of magnitude smaller than the execution frequency of the continuously evaluated stream processing queries. The average durations of the query derivation are 3.578 seconds and 2.968 seconds for the generic DIVIDE queries that result in the stream processing queries that detect toileting and brushing teeth activities, respectively. When comparing these values to the average execution times of the real-time reasoning queries for these activities on the streaming version of the RDFox engine (0.716 seconds and 0.423 seconds), it can be concluded that the durations of the query derivation with DIVIDE are still lower, compared to an order of magnitude (i.e., 10 times) larger than these average query execution times on RDFox.

Considering the design of DIVIDE and the presented evaluation results, it is also clear that **research contribution RCO2 allows validating research hypothesis RH4**: "The methodological design of a semantic IoT platform component that enables privacy by design will let the end user in 100% control about which data abstractions can be sent over the network and which data is not leaving the local environments of the IoT network, while maintaining an overhead to adapt the queries based on changing use case context that is at most 1 order of magnitude (i.e., 10 times) higher than the execution time of semantic queries on equivalent state-of-the-art real-time reasoning set-ups.". This means that the duration of the DIVIDE query derivation is less than 10 times larger than the execution time of the real-time reasoning queries in the best-performing alternative state-of-the-art set-up, while its execution frequency is more than 10 times smaller. This implies that the longer durations of the query derivation in DIVIDE are perfectly acceptable.

In Chapter 4 and Appendix B, the generic design of DIVIDE is also further illustrated by employing it for use case UC3 about the monitoring of headache symptoms and triggers for patients that are diagnosed with a primary headache disorder. For this specific use case, the context-aware queries derived by DIVIDE help in moving towards continuous, semi-autonomous, objective follow-up and classification of primary headache disorders.

The generic design of DIVIDE allows extrapolating the presented findings to employ research contribution RCO2 for other IoT application domains. In general, DIVIDE can be employed for any IoT application that can benefit from deriving (actionable) insights from merging streaming data with domain knowledge and use case context information. More specifically, this means that DIVIDE can be deployed as an additional central component in any semantic IoT platform that uses RSP engines to evaluate continuous queries. It is especially useful when being employed in a heterogeneous IoT network with low-end local processing devices, since its simple queries can be efficiently evaluated on such devices. DIVIDE can automatically deal with changes in use case context by deriving queries on context changes. It allows configuring in generic query templates how both the conditions and window parameters of queries can be instantiated according to the actual use case context. The frequency of changes in use case context should be at least an order of magnitude smaller than the required evaluation frequency of the continuous RSP queries, to compensate for the slightly larger duration of the query derivation process. However, in practice, this requirement is achieved in almost every use case. Due to the automatic reconfiguration of the RSP queries, an end user only needs to configure DIVIDE once upon initialization of the system. Moreover, DIVIDE enables privacy by design, meaning that privacy by design can be built into the application. This can be configured to the required extent for any individual use case, and thus does not limit the use cases for which DIVIDE is considered useful.

Research challenge RCH3: Adaptive configuration and distribution of stream processing queries based on situational context

To address the third research challenge of this dissertation, the semantic IoT platform component DIVIDE is further extended in Chapter 5. This represents research contribution RCO3 of this dissertation. The version of DIVIDE resulting from Chapter 3 is only adaptive to changing use case context. However, situational context is another aspect of the environmental context in which stream processing queries are evaluated in a semantic IoT platform. Therefore, Chapter 5 extends the methodological design of DIVIDE. A local monitor is designed that can monitor various situational context parameters. In its current design, the local monitor contains three individual monitors that monitor (i) networking characteristics, (ii) resource usage of the local stream processing devices, and (iii) data stream properties and real-time query performance of the local stream processing engines. A meta model ontology is designed that allows modeling the monitored information, as well as meta-information about devices, components and the configuration and distribution of the deployed stream processing queries. Using this ontology, local monitoring observations are semantically annotated, aggregated and forwarded to a global monitor that can update the distribution and configuration of stream processing queries in the platform. In concrete, queries can be moved between a local and central RSP engine, and the window parameters of queries can be modified. This is achieved by continuously evaluating global monitor queries on the maintained meta model. Using these queries, end users can define for every individual use case how the situational context should influence the query distribution and configuration. This way, use case specific trade-offs can be automatically balanced and efficient stream reasoning can be achieved. The modular design of the meta model ontology and the subcomponents of DIVIDE ensures that the monitoring of additional situational context properties can easily be added to the design in the future. The extended DIVIDE methodology has also been realized in a first implementation, by building further on the first implementation of DIVIDE that resulted from research contribution RCO2 in Chapter 3.

In Chapter 5, the extended DIVIDE implementation is evaluated on the homecare monitoring use case UC2 introduced in Chapter 3. In the discussed use case scenario, queries are preferably executed as much as possible on the central RSP engine, because the raw data is needed on the central servers to be visualized in dashboards, to motivate decisions, and for post-intervention analysis. This results in global monitor queries that only move the queries locally if the need arises. The evaluations on this use case in Chapter 5 are performed by replaying an anonymous representative part of a real-word dataset of IoT sensor data in a smart home environment, that is the result of a large scale data collection process. This is similar to the evaluations in Chapter 3. In total, the replayed dataset contains on average 186 observations per second.

The evaluation contains two scenarios in which it is shown that DIVIDE can update the query distribution and configuration according to the conditions

and thresholds defined on monitored situational context properties in the global monitor queries. More precisely, the first scenario illustrates how the window size and frequency of queries can be lowered if the query performance on local, low-end devices worsens. In addition, the second scenario shows how queries are moved from the central RSP engine to the local RSP engines whenever the networking capacity decreases and does not allow the efficient forwarding of all raw sensor data over the network.

Considering the design of DIVIDE and the presented results of the evaluations, it is clear that **research contribution RCO3 allows validating research hypothesis RH5**: "The methodological design of a semantic IoT platform component that monitors the situational context will result in an adaptive system that can update the window parameter configuration and distribution (i.e., location) to varying situational context, *precisely* according to use case specific rules and thresholds as defined by the end user, for a realistic local data stream of at least 150 observations per second.".

When extrapolating these findings to employ research contribution RCO3 for other IoT application domains, it is clear that DIVIDE can be used in any IoT application that is deployed in a dynamic environment with varying use case and/or situational context and that has to deal with high-velocity data streams. This is a consequence of the generic design of DIVIDE. Specifically focusing on the varying situational context, DIVIDE is able to deal with variations in any of the situational context properties that are being monitored. To do so, the end user only has to write a semantic query for the global monitor that specifies when and how the configuration and distribution of queries should be updated, using the concepts from the designed meta model ontology. By building further on the modular design of DIVIDE and its implementation, additional situational context properties could be incorporated to support other parameters important to specific IoT application domains.

Research challenge RCH4: Closing the loop by embedding the solutions into a full semantic platform that is efficient & performant

To address the fourth and final challenge of this dissertation, the semantic IoT platform component DIVIDE is embedded into a full semantic platform in Chapter 6. This represents research contribution RCO4 of this dissertation. The chapter presents a distributed reference architecture that can be mapped to the generic design of the cascading reasoning framework designed in Chapter 2. The architecture brings together different tools built upon Semantic Web technologies in a performant and easily configurable manner. The architecture allows designing flexible, data-driven services for every individual use case that operate on the data abstractions and insights that are generated by the stream reasoning queries managed by DIVIDE. In addition, a semantic workflow engine is included that can compose dynamic workflows based on a semantic description of required functionality and quality requirements of the workflow. This semantic workflow engine can also automatically detect conflicts between

workflows. Resulting knowledge generated by the semantic services and through the workflows can update the use case context, which in turn triggers DIVIDE to adaptively update the context-aware queries. This way, chaining the different building blocks in the cascading reasoning architecture helps closing the feedback loop in IoT applications. In general, by using Semantic Web technologies, declarative solutions can be built with the platform: the desired actions (e.g., instructions to generate semantic data, queries, services, or workflow steps) can be provided by system installers in a declarative way, independently of their exact implementation.

In Chapter 6, the generic reference architecture of the semantic platform is applied to the healthcare application domain. This way, it zooms in on a semantic healthcare platform of which the different building blocks can be leveraged to optimize continuous (home)care provisioning. To this end, a use case demonstrator of such a healthcare platform is built for the homecare monitoring use case UC2. In this chapter, the focus of this use case is on smart personalized monitoring of homecare patients based on a patient's medical profile, and the construction of workflows that represent treatment plans to the diagnoses in the patient's profile. The demonstrator shows the possible role of the different building blocks through a realistic scenario. It highlights how DIVIDE can be embedded into a healthcare platform and allows creating a feedback loop. Moreover, it demonstrates how the semantic workflow engine can help improving the coordination of workflows across organizations and stakeholders involved in the patient's caregiving.

Considering the design of the reference architecture and evaluating its application on a continuous homecare use case in the healthcare domain, it can be concluded that **research contribution RCO4 allows validating research hypothesis RH6**: "A semantic IoT platform component that adaptively manages and configures queries according to varying environmental context, can be embedded in a semantic platform with other semantic components that define and construct data-driven semantic services and cross-organizational semantic workflows. Put together, the resulting cascading reasoning architecture can be leveraged to optimize relevant IoT use cases.".

The generic design of the reference architecture and its semantic building blocks allows employing the semantic platform for other IoT application domains as well. In general, any IoT application that works with streaming data could follow the cascading reasoning design of the platform. The modular design of the architecture implies that the building blocks can be selected and left out as required, on a per use case basis. For example, if no concept of workflow is relevant to the use case, it is perfectly possible to omit the semantic workflow engine AMADEUS from the platform design. Moreover, the modular design also allows integrating the platform more easily with existing components already in place in certain applications. The advantages of adding DIVIDE to the design of the platform for a given use case include closing the feedback loop between possible services or workflows further in the cascading reasoning pipeline and the first components in the chain.

7.2 Open challenges and future directions

This dissertation has focused on important challenges in the domain of adaptive and performant stream reasoning, with a focus on evaluating the contributions on use cases from the healthcare application domain. Nevertheless, different challenges remain, offering the possibility to further build on the presented research. To conclude this dissertation, this section discusses a selection of the most important open challenges and suggests possible future work directions to solve them.

7.2.1 Integrating the dynamic deployment of stream reasoners across the network

In this dissertation, adaptiveness to environmental context has been introduced in stream reasoning systems. To this end, DIVIDE was designed. By using DIVIDE as an additional component in a semantic IoT platform, the configuration and distribution of the queries on the platform's stream reasoning components is no longer static, but adaptive to changing environmental context. This increases the dynamic aspect of platforms: queries can be updated over time, and they can move between local and central stream reasoning components. Nevertheless, in the current set-up, the configuration and deployment of the active stream reasoning *components* in the IoT platform is still static. This implies that a fixed chain of cascading reasoning components is manually created when configuring the system. Therefore, the opportunity remains to further increase the dynamic aspect of the set-up in the future. More specifically, adaptive algorithms could be designed and incorporated in the global monitor component of DIVIDE that dynamically deploy stream reasoning engines across the IoT network. To this end, the design of DIVIDE should be further extended to allow dealing with a *dynamic* pipeline of stream reasoning components in the cascading reasoning architecture, that can vary over time. Depending on the monitored environmental context information, new stream reasoning engines could then be deployed or existing ones could be removed from the platform. This way, the distribution of queries across the available engines would become more dynamic too.

To optimally design the algorithms that increase the dynamic nature of the distribution of stream reasoning engines and queries, the currently monitored environmental context could be used. In addition, these algorithms could further consider the overall scalability of the system and the performance of the stream reasoners on a global and local level. This way, these aspects could be further improved through the additional dynamic deployment of stream reasoning engines. Similarly, other requirements, such as properly dealing with a total loss of network connectivity, could be considered as well. Importantly, the designed algorithms should optimally balance default distribution strategies based on general requirements, such as performance and scalability, with use case specific requirements that can currently already

be incorporated through the global monitor queries. To this end, it should also be researched how these algorithms could be integrated with existing platforms that support the distribution of services in fog computing architectures according to various requirements [6, 7]. Moreover, additional individual monitors could be implemented and deployed to enrich the input of these algorithms, such as a monitor of the energy consumption of devices.

To further extend the dynamic nature of the distribution of reasoning tasks, the algorithms could also incorporate intelligence with respect to the distribution of domain knowledge and use case context information, which is used for the semantic reasoning and evaluation of queries, across the network. This intelligence should take into account existing requirements, such as data privacy and local autonomy, but could also consider functional requirements, such as considerations about which data often needs to be considered together. In addition, such algorithms could also take the duplication of certain data into account. Importantly, by moving around the data, the algorithms should ensure that the correctness of the processing is still guaranteed, and that no data is lost or unconsidered.

A final interesting pathway for future research in this area is the integration of algorithms that try to predict the consequences of updating a distribution strategy, *before* the distribution is actually updated. This would be especially useful when the distribution decided by the designed set of algorithms considers stream reasoning engines, queries *and* data. In such cases, updating the distribution would become less straightforward and would imply a certain cost. By predicting these consequences in terms of performance or other requirements using for example data-driven machine learning techniques, distribution changes could be made more carefully [8]. Similarly, algorithms could be designed that try to predict future situational context, to take this into account in the distribution algorithms. However, having the data to design these different algorithms would be a big challenge and might therefore even become a practical burden, since gathering a sufficiently large dataset about all possible scenarios across the varying environment could become very complicated. Therefore, a very interesting approach in this area would be the investigation of reinforcement learning algorithms that could learn by themselves over time how the full system should organize itself [9]. Such algorithms avoid the need of obtaining a dataset upfront and are thus ideally suited for very dynamic environments. In addition, they generalize well as they should not start from scratch when being applied to new environments or use cases: they already know how to deal with common situations, and can adapt themselves over time to learn use case specific situations. The main difficulty in designing these algorithms is in the design of the state representation, reward or cost function, and the decision-making policy of distribution changes. In general, these are examples where data-driven algorithms could be used together with knowledge-driven applications, showing the promising potential of hybrid AI approaches. Other such examples include the data-driven learning of certain domain knowledge.

7.2.2 Improving the user-friendliness of the solutions

To ensure that the solutions presented in the different contributions of this dissertation could be used more easily by installers of IoT applications in domains such as healthcare, additional efforts should be taken to improve their user-friendliness. The resulting user interface (UI) tools would extend the set of available UIs for the different semantic building blocks in the reference architecture of the generic semantic platform as presented in Chapter 6. This way, the reduced complexity for installers would allow increasing the adoption of this dissertation's solutions at people who are not necessarily an expert in the domains of Information and Communication Technology (ICT) or semantics, e.g., healthcare professionals.

To properly set up and configure the cascading reasoning framework presented in Chapter 2, a UI should be designed. This configuration is especially tedious when complex networks of stream reasoning components are designed, since they should be correctly chained. To use DIVIDE in a semantic IoT platform, only the initial static configuration of devices and stream reasoning engines, and the DIVIDE query templates should be provided by the installer. For the latter, Chapter 3 has already discussed how existing stream reasoning queries can be easily reused in the configuration for this. Nevertheless, these configurations cannot yet be made with a UI. Especially for the definition of the global monitor queries to update the query configuration and distribution based on the monitored situational context with DIVIDE, extra efforts are required to let end users configure which situational context properties should alter the query configuration and distribution, and how. This is important since this inherently should not require knowledge of Semantic Web technologies: these queries are actually semantic translations of certain actuation rules, which capture the use case specific requirements. Chapter 5 has already suggested a user-friendly grammar to specify these rules, but UI tooling is still required for this. These tools could also automatically suggest relevant rules, and take into account different behavior of the rules for different queries (e.g., based on window parameters or query conditions) or people (e.g., based on user profile) by incorporating priorities into the methodological design of DIVIDE. Finally, similarly to DIVIDE, no UI tool is available yet for the semantic workflow engine discussed in Chapter 6.

To ensure that proper UIs are built, co-design sessions and user workshops should be organized with the actual installers that will be configuring the semantic IoT platforms [10]. To this end, the full semantic platform should be applied to various applications and use cases in the relevant IoT application domain, e.g., healthcare. This way, the needs and requirements for installers can be properly captured, as well as how the UI tools could optimally support the configuration and installation of the platform for different use cases in this domain that can have different key requirements. As such, the resulting UI tools could optimally support the further improvement of IoT applications in domains like healthcare.

7.2.3 Further addressing the privacy and security requirements

Privacy and security are two very important requirements when considering IoT applications. This is true for multiple application domains. Healthcare is an important example of this, as privacy of patient data is key, especially when considering IoT in healthcare [11]. The privacy requirement is addressed in this dissertation, but open challenges remain. Therefore, before deploying the solutions presented in this dissertation in a real-life setting, additional measures should be researched and integrated to improve the privacy and security of the system.

As presented in Chapter 3, the designed DIVIDE component *enables* privacy by design [12]. This means that it allows its end users to build privacy by design in the application. By definition, privacy by design states that privacy must be incorporated into data systems and technologies, by default. As a guideline on how to achieve privacy by design, it consists of seven foundational principles. By embedding DIVIDE into the cascading reasoning reference architecture presented in this dissertation, multiple of those principles are addressed to some extent, helping end users to design applications with the privacy of user data in mind. Privacy by design also is a key principle of the General Data Protection Regulation (GDPR) of the European Union, which further emphasizes the importance of properly considering it. This is of course also true in general for GDPR.

Zooming in on the individual principles of privacy by design, it is clear that the methodological design of DIVIDE allows system installers to embed privacy into the design of software applications. Moreover, by letting system installers define generic DIVIDE query templates, they can proactively ensure that sensitive data stays local. This allows them to work with strong default privacy measures in the design of these templates, in which the possible data abstractions that can leave the local environments are visible and transparent. From a methodological point of view, this allows for a user-centric approach where end users remain in control of their privacy. However, several improvements are possible in how DIVIDE addresses privacy by design. A very powerful approach to do so is by incorporating privacy into the design of UI tools to configure the cascading reasoning framework and DIVIDE, of which the open challenges and future directions are discussed in Section 7.2.2. This would ensure that the management of privacy becomes more user-centric, it would further increase the visibility and transparency of what is happening with the user's data, and it would allow installers to translate maximal privacy into the default DIVIDE query templates of the system. Complying with the visibility & transparency and user-centric principles of privacy by design is especially important to also ensure that users are willing to share their personal data with applications, as this is essential to the designed solutions. Moreover, it is important to consider privacy as a strong requirement when studying the dynamic deployment of stream reasoners across the network, which is discussed in Section 7.2.1. This is required since the increased dynamism in automatically updating the distribution of stream reasoning engines, queries *and* data could

decrease the visibility and transparency of the privacy solutions and might conflict with default privacy settings put in place by the installers. Therefore, jointly considering both open challenges is of key importance. Finally, end-to-end security is another foundational principle of privacy by design, which is not yet considered in this dissertation and should thus still be addressed.

Considering how DIVIDE enables privacy by design, it is clear that the system installers have much responsibility in optimally addressing privacy in IoT applications. This highlights the need to involve privacy experts to teach installers, and possibly end users if UI tools allow them to further control data privacy of themselves or other users as well, about how to optimally manage data privacy, possible risks, and more. This might be an additional challenge for smaller organizations that do not always have the budget to consult these experts.

In general, apart from the specific principles of privacy by design, it is important to note that using the cascading reasoning framework with DIVIDE in a distributed, decentralized system is no guarantee for privacy as such. In other words, DIVIDE *enables* privacy by design, but it does not *guarantee* privacy. Hence, a set of additional privacy measures has to be put in place. Classic measures could be incorporated, such as strong cryptography and access control mechanisms. In addition, other privacy solutions often depend on use case specific requirements. Nonetheless, a broad research field focuses on designing privacy solutions for specific application domains. For example, in healthcare, much research exists about privacy solutions for healthcare applications in general and the integration of IoT in healthcare in specific [13]. Hence, existing privacy solutions could be leveraged and integrated into the cascading reasoning framework with DIVIDE and the other semantic building blocks. This has been made more easily possible through the modular design of the cascading reasoning architecture and its components. In summary, in its current state, it remains the responsibility of the users of the framework and DIVIDE to research, implement and integrate additional privacy measures into the design of applications, to achieve optimal privacy preservation.

It should be noted that the core focus of the discussion on privacy and privacy by design for the solutions in this dissertation is on considering the privacy of the communication of data. Of course, this does not omit the need to consider the privacy of data on the devices themselves. This is especially important in use cases where the (raw) data will also be stored locally, for example for visualization purposes on local dashboards in homecare, or for using it as training or input data for data-driven algorithms that are used in the system (e.g., algorithms to derive predicted events considered in the headache monitoring use case UC3 discussed in Chapter 4). Considering the provision of privacy guarantees on local devices, it is important to mention that the usage of multiple devices in a distributed network also complicates privacy. For example, one could argue that complying with some of the principles of privacy by design and securing the privacy of user data on a single high-end device in a centralized approach

is easier compared to doing so on a larger set of low-end devices in a decentralized approach. This is definitely true, since the data can be spread out over devices and the lack of resources on low-end devices restricts the complexity of privacy solutions that can be integrated. Nevertheless, this is a requirement one has to deal with, given the existing conditions of real-world IoT networks and their devices.

Specifically zooming in on security, multiple security systems and frameworks are being researched, for example in the domain of healthcare and the IoT [13], similarly to privacy in general. Currently, the solutions presented in this dissertation do not specifically address security and their usage thus does not guarantee any additional security to the system. Therefore, leveraging existing security research and integrating it into the presented solutions is also key when deploying them in real-life application settings. As mentioned earlier, providing end-to-end security through the life cycle of data is also one of the seven foundational principles of privacy by design. This highlights the relevance of considering security jointly with privacy, in terms of the software, the communication, and the data itself.

7.2.4 Integrating with Solid

Recently, there has been an uplift of Solid [14, 15]. The core vision of the Solid project consists of independent, decentralized, personal data stores which are called Pods. By letting users store their data securely in their own Pods, services can be decoupled from the data and users can obtain full fine-grained control about where and how their data is used & shared with applications and other people. Solid offers multiple additional benefits, such as having easy access to semantic data, having a single source of truth, enabling data reuse by multiple apps, and complying more easily with regulations such as GDPR. Specifically considering healthcare, the Solid project forms an interesting opportunity, because its vision aligns with the key importance of privacy and end user data control in healthcare solutions. The same is also true for other IoT application domains that deal with sensitive user data. In this context, it would be important to consider this challenge jointly with the privacy challenge discussed in Section 7.2.3. To examine the opportunity provided by Solid, future research should investigate how the adaptive, cascading reasoning framework designed in this dissertation could be integrated with the Solid project. To this end, the cascading reasoning framework should be aligned with the decentralized vision of Solid, in order to optimize the resulting system according to the Solid specifications and infrastructure.

7.2.5 Formalizing semantic modeling decisions

In this dissertation, several semantic components have been designed, such as the generic cascading reasoning framework with its individual stream reasoning components, and DIVIDE. Since this dissertation has considered other aspects when discussing the methodological design and modeling decisions of these components

across its different chapters, future work could also look into the formalization of the semantic modeling decisions in the components' design. In this context, a distinction should be made between the stream reasoning components of the cascading reasoning framework, and DIVIDE.

Considering the stream reasoning components of the cascading reasoning framework, this dissertation has mainly focused on their architectural design and how the requirements of the components have been taken into account for that, and on practical aspects from considering real-world use cases. However, several aspects of the semantic modeling decisions in the design of the components of the cascading reasoning framework could be formalized. This includes the used definition of a semantic data stream and the modeling of time, the consideration of time in a single component and across different components in a cascading reasoning pipeline, the semantics of stream merging and continuous reasoning & query processing across the cascading reasoning pipeline, and the semantics of the continuous queries themselves (e.g., formally defining how windows are expressed, and when the query results are computed). Moreover, this also includes possible assumptions with respect to the input data, query evaluation and semantic reasoning. When considering the formalization, it would be important to make a distinction between actual RSP engines (e.g., the RSPS component in the generic cascading reasoning architecture presented in Chapter 2), of which the queries can be managed by DIVIDE, and other stream reasoning components (e.g., the LRS and BRS components in the generic cascading reasoning architecture). For the former, RSP-QL could be employed to perform this formalization [16]. This is a reference model that unifies the semantics of existing RSP approaches, and it is also considered in the definition of generic RSP queries with DIVIDE.

Considering the semantic modeling decisions of DIVIDE, one could look into the formalization of the DIVIDE query derivation process. Currently, the details of this process and the preceding DIVIDE initialization have been discussed in terms of which steps are applied and which semantic rules, queries and/or reasoning operations are involved. In addition, the formal aspects of these steps could be looked into as well. Moreover, the impact of integrating adaptiveness to situational context could also be formally assessed.

Formalizing the modeling decisions and operation modes of the semantic components in this dissertation would help with studying to what extent the presented solutions can assure correctness or provide guarantees with respect to the accuracy of the insights, decisions and actions derived by the semantic components. Moreover, it would allow semantic experts and installers of other use cases in healthcare and other IoT application domains to formally assess when and how the solutions designed in this dissertation could be used.

7.2.6 Evaluating and extending the presented generic solutions for other IoT application domains

The evaluations and validation of the solutions presented in this dissertation are focused on the healthcare domain, which is an important IoT application domain. Important requirements of applications in the healthcare domain include those considered by this dissertation, such as responsiveness, local autonomy, privacy, level of adaptiveness, automation, and configurability. Nevertheless, the solutions of this dissertation mostly have a generic design, technically allowing them to be also applied to other IoT application domains that have similar requirements. As highlighted in the beginning of this dissertation in Chapter 1, the IoT has transformed many other application domains that involve human-machine interaction. Many of them do indeed also present many of the considered requirements. An example of such a domain is the challenging domain of smart cities, where smart technological solutions are being designed to address sustainable living and increase the comfort, productivity and overall quality-of-life of citizens [17]. Many other IoT application domains exist as well, such as agriculture, smart home and automation, energy, logistics, and others [18]. To use the presented solutions in these other application domains, representative use cases have to be chosen to map the reference cascading reasoning architecture to. This should happen in close collaboration with experts in the considered application domain. This would then allow further investigating how the generic design and configurability of a semantic IoT platform employing the cascading reasoning framework and DIVIDE, can be further extended to integrate additional, potentially different domain-specific requirements as well.

7.3 Closing words

"If you think that the internet has changed your life, think again. The Internet of Things is about to change it all over again!". To conclude this dissertation, I feel that it makes sense to repeat the opening quote of the dissertation about the impact of the IoT on our society and our lives. Not the least in healthcare, the IoT has opened the gates to further optimize and improve continuous care. To this end, this dissertation has tried to make the semantic reasoning on IoT data streams more adaptive, by designing different solutions and validating them on several healthcare use cases. In this process, various other requirements have been considered as well, such as performance, responsiveness, local autonomy, and privacy. As will ever be the case, open challenges remain, which will hopefully be tackled in the upcoming years. Nevertheless, I sincerely hope that I did my bit in contributing to this exciting future of the IoT, especially in healthcare.

References

[1] R. I. Goleva, N. M. Garcia, C. X. Mavromoustakis, C. Dobre, G. Mastorakis, R. Stainov, I. Chorbev, and V. Trajkovik. *Chapter 8 – AAL and ELE Platform Architecture*. In C. Dobre, C. Mavromoustakis, N. Garcia, R. Goleva, and G. Mastorakis, editors, Ambient Assisted Living and Enhanced Living Environments: Principles, Technologies and Control, pages 171–209. Butterworth-Heinemann, 2017. doi:10.1016/B978-0-12-805195-5.00008-9.

[2] F. Ongenae, P. Duysburgh, N. Sulmon, M. Verstraete, L. Bleumers, S. De Zutter, S. Verstichel, A. Ackaert, A. Jacobs, and F. De Turck. *An ontology co-design method for the co-creation of a continuous care ontology*. Applied Ontology, 9(1):27–64, 2014. doi:10.3233/AO-140131.

[3] P. Bonte, F. Ongenae, J. Schaballie, B. De Meester, D. Arndt, W. Dereuddre, J. Bhatti, S. Verstichel, R. Verborgh, R. Van de Walle, E. Mannens, and F. De Turck. *Evaluation and optimized usage of OWL 2 reasoners in an event-based eHealth context*. In Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation (ORE-2015), co-located with the 28th International Workshop on Description Logics (DL 2015), pages 1–7, 2015. Available from: https://ceur-ws.org/Vol-1387/paper_6.pdf.

[4] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. *C-SPARQL: a continuous query language for RDF data streams*. International Journal of Semantic Computing, 4(1):3–25, 2010. doi:10.1142/S1793351X10000936.

[5] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. *RDFox: A highly-scalable RDF store*. In The Semantic Web - ISWC 2015: Proceedings, Part II of the 14th International Semantic Web Conference, pages 3–20, Cham, Switzerland, 2015. Springer. doi:10.1007/978-3-319-25010-6_1.

[6] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. *Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications*. In 2019 IEEE Conference on Network Softwarization (NetSoft), pages 351–359, 2019. doi:10.1109/NETSOFT.2019.8806671.

[7] M. Sebrechts, B. Volckaert, F. De Turck, K. Yangy, and M. AL-Naday. *Fog Native Architecture: Intent-Based Workflows to Take Cloud Native Towards the Edge*. IEEE Communications Magazine, 60(8):44–50, 2022. doi:10.1109/MCOM.003.2101075.

[8] S. Verma and A. Bala. *Auto-scaling techniques for IoT-based cloud applications: a review*. Cluster Computing, 24(3):2425–2459, 2021. doi:10.1007/s10586-021-03265-9.

[9] H. Tran-Dang, S. Bhardwaj, T. Rahim, A. Musaddiq, and D.-S. Kim. *Reinforcement learning based resource management for fog computing environment: Literature review, challenges, and open issues*. Journal of Communications and Networks, 24(1):83–98, 2022. doi:10.23919/JCN.2021.000041.

[10] J. C. Bastien. *Usability testing: a review of some methodological and technical aspects of the method*. International Journal of Medical Informatics, 79(4), 2010. doi:10.1016/j.ijmedinf.2008.12.004.

[11] A. Chacko and T. Hayajneh. *Security and privacy issues with IoT in healthcare*. EAI Endorsed Transactions on Pervasive Health and Technology, 4(14), 2018. doi:10.4108/eai.13-7-2018.155079.

[12] A. Cavoukian. *Privacy by design*, 2009. Accessed: 2022-09-25. Available from: https://www.ipc.on.ca/wp-content/uploads/Resources/7foundationalprinciples.pdf.

[13] J. J. Hathaliya and S. Tanwar. *An exhaustive survey on security and privacy issues in Healthcare 4.0*. Computer Communications, 153:311–335, 2020. doi:10.1016/j.comcom.2020.02.018.

[14] R. Verborgh. *Re-decentralizing the Web, for good this time*. In O. Seneviratne and J. Hendler, editors, Linking the World's Information: A Collection of Essays on the Work of Sir Tim Berners-Lee. ACM, 2022. Available from: https://ruben.verborgh.org/articles/redecentralizing-the-web/.

[15] *Solid*. Accessed: 2023-04-14. Available from: https://solidproject.org/.

[16] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte, and O. Corcho. *RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems*. International Journal on Semantic Web and Information Systems (IJSWIS), 10(4):17–44, 2014. Available from: https://dl.acm.org/doi/10.5555/2795081.2795083.

[17] A. S. Syed, D. Sierra-Sosa, A. Kumar, and A. Elmaghraby. *IoT in Smart Cities: A Survey of Technologies, Practices and Challenges*. Smart Cities, 4(2):429–475, 2021. doi:10.3390/smartcities4020024.

[18] Y. Perwej, K. Haq, F. Parwej, M. Mumdouh, and M. Hassan. *The Internet of Things (IoT) and its application domains*. International Journal of Computer Applications, 182(49), 2019. doi:10.5120/IJCA2019918763.

A

Personalized Real-Time Monitoring of Amateur Cyclists on Low-End Devices: Proof-of-Concept & Performance Evaluation

In Chapter 2, a generic cascading reasoning framework for healthcare and other IoT applications was proposed to solve the issues in existing centralized solutions. This framework exploits the availability of the heterogeneous low-end local & edge devices in an IoT network: it uses these devices to host the first components in the stream reasoning pipeline of the cascading reasoning architecture. This appendix specifically focuses on these local processing components for use case UC4. This use case is about the personalized real-time monitoring of amateur cyclists, and is also linked to healthcare. The appendix presents a Proof-of-Concept of a real-time feedback system deployed on a Raspberry Pi device that uses semantics and stream reasoning to give real-time feedback to cyclists about their heart rate and heart rate training zones, personalized according to the cyclists' profile. This real-time feedback system can thus be considered a local stream processing component in the cascading reasoning framework presented in Chapter 2. In this appendix, the performance of the resulting Proof-of-Concept on a low-end device is also evaluated.

⋆⋆⋆

M. De Brouwer, F. Ongenae, G. Daneels, E. Municio, J. Famaey, S. Latré, and F. De Turck

Published in the Companion Proceedings of The World Wide Web Conference 2018, April 2018.

Abstract
Enabling real-time collection and analysis of cyclist sensor data could allow amateur cyclists to continuously monitor themselves, receive personalized feedback on their performance, and communicate with each other during cycling events. Semantic Web technologies enable intelligent consolidation of all available context and sensor data. Stream reasoning techniques allow to perform advanced processing tasks by correlating the consolidated data to enable personalized and context-aware real-time feedback. In this appendix, these technologies are leveraged and evaluated to design a Proof-of-Concept application of a personalized real-time feedback platform for amateur cyclists. Real-time feedback about the user's heart rate and heart rate training zones is given through a web application. The performance and scalability of the platform is evaluated on a Raspberry Pi. This shows the potential of the framework to be used in real-life cycling by small groups of amateur cyclists, who can only access low-end devices during events and training.

A.1 Introduction

In recent years, the importance of using data to take strategic decisions in sports, both for professional events and amateur training, has significantly increased [1]. This monitoring is especially the case in cycling [2, 3]. Many cyclists are riding with a series of sensors measuring aspects such as heart rate, power, location, speed, altitude and cadence. However, existing cycling training apps, such as Strava[1], do almost all data reporting and analysis offline as a post-processing step.

Real-time collection and analysis can therefore bring important innovations in the cycling world, especially for amateur cyclists, who typically do not have the same resources as professional cycling teams. Enabling this can allow them to continuously monitor themselves, receive personalized feedback on how they are performing, and communicate with others during amateur cycling events, such as the Tour of Flanders for amateurs. To make the feedback more valuable for each rider, it should be personalized. Many different parameters define the physiological profile of a rider. For example, the resting and maximum heart rate define the training zone boundaries, which means other feedback may apply for different riders having the same heart rate. To be able to act immediately upon received feedback, its real-time aspect is important. Depending on the parameter, real-time requirements can differ. For example, power is much more oscillating than heart rate, meaning power feedback can only be considered real-time when updates occur at least every second, whereas for heart rate this can be up to 5 seconds.

[1] https://www.strava.com

To achieve the real-time data collection and analysis, two major technological innovations are researched. First, cycling events often take place at remote areas without any cellular connectivity. This is a challenge for data collection. Therefore, a novel IoT platform for challenging environments is described in this appendix [4]. It serves as a mobile network layer with the bikes as nodes, allowing to send real-time sensor information from cyclist to cyclist, without the need of any Internet connection.

Second, a data analysis layer is researched, allowing for intelligent real-time feedback before and during sports events, more specifically amateur cycling events. To make this feedback personalized and context-aware, the system should be able to take into account context information, such as rider profiles, route information, weather etc. Given all available context information and the observations of the different sensors, intelligent consolidation and analysis of this data is required. As this data is often heterogeneous, semantics are the ideal approach to tackle this issue [5]. Ontologies can be used to model the data and their relationships and properties. Stream reasoning techniques then allow to perform advanced processing tasks that enable to design personalized and context-aware real-time feedback, by mapping the continuous data streams on the available background knowledge modeled in an ontology [6]. For example, an incoming heart rate observation of a sensor can be linked to the corresponding rider, allowing to retrieve his profile information and determine his current heart rate training zone and associated feedback depending on this person's boundaries. An added advantage of semantics is the usage of generic queries, allowing context data, e.g., new sensor types, and queries to be added to a running system, without the need for adaptations.

Within the imec ICON project CONAMO (CONtinuous Athlete MOnitoring)[2], a Proof-of-Concept (PoC) application was realized, demonstrating what can be achieved by implementing a real-time feedback platform for cyclists using the two technological advances discussed above. To allow adoption by amateur cyclists, the real-time feedback system should be able to run on a low-end device, e.g., the GPS device or smartphone used on their bike. Therefore, the performance and scalability of the designed platform is evaluated on a Raspberry Pi.

The outline of this appendix is as follows. Section A.2 presents related work in the area of stream reasoning, semantics in sports, and IoT data collection. Section A.3 details the PoC use case, while Section A.4 presents its architecture set-up. The IoT platform for real-time data collection is described in Section A.5, whereas Section A.6 discusses the real-time feedback platform itself. In Section A.7 and A.8, the performance of the feedback system is evaluated on a Raspberry Pi. Finally, Section A.9 and A.10 discuss and conclude the main findings.

[2]https://www.imec-int.com/nl/imec-icon/research-portfolio/conamo

A.2 Related work

A.2.1 Stream reasoning

Data Stream Management Systems (DSMS) & Complex Event Processing (CEP) allow to query homogeneous streaming data structured according to a fixed data model [7]. In contrast to Semantic Web reasoners, DSMS & CEP are unable to deal with heterogeneous data sources and lack support for the integration of domain knowledge. To bridge this gap, stream reasoning focuses on the scalable and efficient adoption of Semantic Web technologies for streaming data [6]. In the past years, several RDF Stream Processing (RSP) engines have been developed [8], of which C-SPARQL [9] and CQELS [10] are the most well-known. They define a window on top of the stream and allow the registration of semantic queries which are continuously evaluated as data flows through the window. These RSP engines can thus filter & query a continuous flow of data, provide real-time answers to the registered queries and support the integration of domain knowledge into the querying process [6]. C-SPARQL enables RDFS reasoning, whereas CQELS does not include any reasoning support. Some preliminary research has been done on publishing RDF streams from low-end devices [11].

A.2.2 Semantic technologies in sports

Some preliminary research has been done about the adoption of semantic technologies for sports. The most advanced of these endeavors is the research conducted in context of the Lifewear ITEA Project [12, 13]. In this research, a wireless sensor network was designed to model sporters who are performing weight-lifting exercises inside a gymnasium. Semantic reasoning is employed to give feedback on their current training schema adherence and to generate alarms when they are taking their physical exercises to a dangerous level. Other endeavors are limited to the semantic annotation and retrieval of sports information [14, 15].

A.2.3 IoT platform for challenging environments

Traditional IoT platforms are based on wireless technologies that are highly dependent on a dense infrastructure of interconnected base stations, and are therefore not suitable for rural, remote and challenging areas with low population density [16–18]. To extend the IoT application to these environments, a number of solutions have been proposed [19–21], using the traditional mobile networks (e.g., GPRS, LTE), NB-IoT [22], LoRa, or SIGFOX to interconnect the IoT devices with a high-speed back-haul network. These infrastructure-based approaches have the disadvantage that deploying and maintaining base stations is expensive and the IoT devices can only function in the presence of a base station. Infrastructure-less approaches on the other

hand [4, 23, 24] can work locally without the need to be connected to the Internet. They are based in low-power multi-hop Wireless Sensor Network (WSN) technologies such as DASH7, Zigbee, and 6TiSCH.

A.3 Use case scenario

The goal is to give real-time personalized feedback to amateur cyclists on their heart rate and heart rate training zone they are currently in. This helps riders to perform the most efficient training instead of over or under training. Important is the personalized approach: all feedback should be adapted to the profile of the rider.

For cyclists, seven heart rate training zones can be distinguished[3]: (1) recovery, (2) long slow distance (LSD), (3) extensive endurance, (4) tempo endurance, (5) block training, (6) extensive interval, (7) anaerobic. For each individual, the boundaries between these training zones can be different. To be able to personalize the feedback, it is therefore important to determine these boundaries based on some profile information.

Based on the resting heart rate and maximum heart rate of a person, the upper bounds of the different training zones can be determined by applying the Karvonen formula [25]:

$$UB_{tz} = intensity_{tz} \times (HR_{max} - HR_{rest}) + HR_{rest}$$

The intensity values for the different training zones are 0.60, 0.64, 0.70, 0.78, 0.84, 0.89 and 1.00 respectively. To determine the resting and maximum heart rate of a person, one should ideally execute professional lab tests, which are not always feasible or realistic for amateur cyclists. Therefore, expertise rules of thumb exist, allowing to determine realistic default values from profile parameters:

- Resting heart rate: this heart rate depends on the level of sportiveness of the person. For a low level of sportiveness, this value is around 65. Similarly, this value is 55 for a medium level and 50 for a high level.

- Maximum heart rate: this value typically decreases with age, and can be estimated as $220 - age$.

By using these expertise rules and the Karvonen formula, a rider's heart rate training zone boundaries can be determined by solely requesting the person's age and level of sportiveness. If a person knows his own maximum and/or resting heart rate himself, these values can of course be used instead of the estimated ones.

[3]Provided by Energy Lab, partner of the CONAMO project.

A.4 Architecture set-up

Figure A.1 visualizes the PoC set-up with two riders[4]. It consists of two parts: the IoT platform for challenging environments (see Section A.5) and the real-time feedback system (see Section A.6).

For personalization purposes, a mobile Android app has been developed, requesting the rider to fill in some profile information before starting the PoC. An Android smartphone running this app is connected to each bike. For reasons explained in Section A.3, date of birth, level of sportiveness and, if known, resting heart rate and maximum heart rate are requested. Name & gender are asked to allow unique identification and later comparison with gender-specific benchmarks. All profile information is stored as context data in the feedback system, to avoid that it needs to be entered every time. Screenshots of the app questions are shown in Figure A.2.

Each rider is wearing a heart rate sensor, measuring the heart rate approximately every second. These observations are sent over different stages and technologies to the real-time feedback system in the back-end, where a stream reasoning framework is deployed. The real-time feedback is shown to the user on a screen, which can for example be on a GPS device or smartphone on the bike or installed in the car, using a locally running web application.

The feedback system is installed on a Raspberry Pi 3, Model B. This low-end device is chosen because it acts as a simple Linux computer, allowing any system running on a laptop to be transported easily. This is for example not possible on an Android tablet.

In Figure A.3, a screenshot of the web application for the real-time feedback system for one rider is shown. Feedback is given about the heart rate, corresponding training zone, and relative training zone distribution for the duration of the ride. When the rider stops cycling, the real-time feedback can be stopped, showing a final training zone distribution with feedback about all training zones. Visual colored feedback (green, orange or red) is given to indicate the intensity of the current effort in relation to a person's aerobic and anaerobic threshold.

A.5 IoT platform for challenging environments

To support real-time collection of sensor data, an IoT platform is proposed that can reliably disseminate data in various dynamic and challenging environments [4]. This platform is shown in the left side of Figure A.1. It is a hybrid solution that combines the advantages of infrastructure-based and infrastructure-less low-power connectivity for IoT in a single multi-modal platform. It uses a variety of Low-power Wireless Personal Area Network (LoWPAN) technologies (e.g., BLE and ANT+) to connect to different sensors that monitor the entity (i.e., the rider), while it also employs an ad-hoc, long-range and multi-hop WSN that reliably disseminates the monitored

[4]It should be noted that the system works with an arbitrary number of riders.

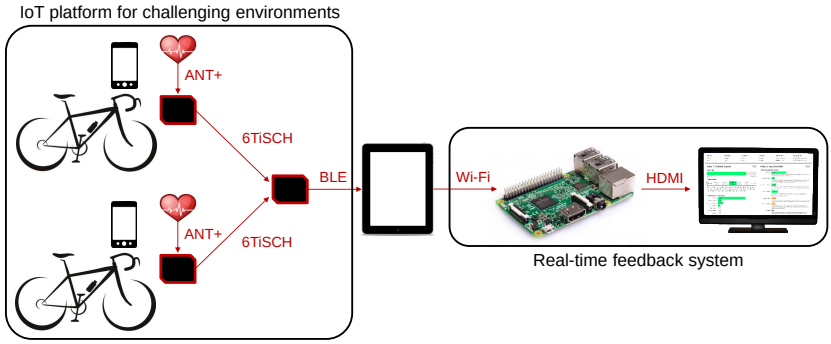


Figure A.1: PoC set-up

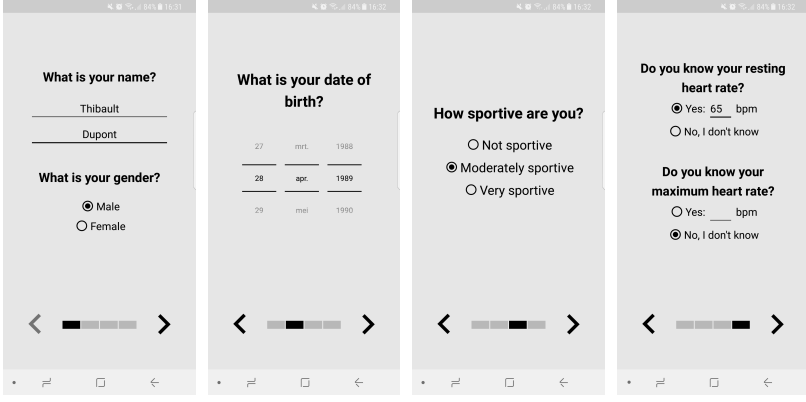


Figure A.2: Screenshots of the Android app requesting a rider's profile information, for feedback personalization

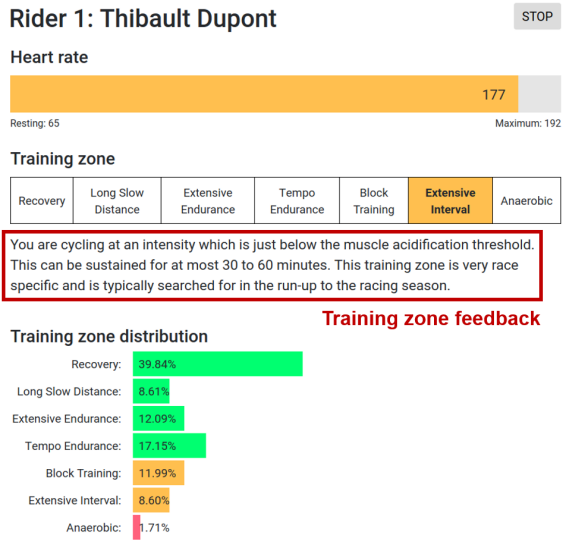


Figure A.3: Screenshot of the web app for the real-time feedback system, shown for only one rider

data to the network sink in real-time. The proposed multi-modal platform provides a highly versatile IoT solution for challenging environments compared to state of the art solutions, with infrastructure-less data dissemination, sporadic disruption-tolerant Internet uplink, and support for heterogeneous wireless sensors and actuators. To allow comprehension of the overall PoC, a short description of the device functionality and data dissemination process of the platform is provided in this section. More details can be found in Daneels et al. [4].

A.5.1 Device functionality

A platform node will collect data from the riders using the sensors it connects to, i.e., it connects to the heart rate monitor of the rider as shown in Figure A.1. The captured data is periodically forwarded into the network to reach the network sink, where it can be analyzed. Being part of a multi-hop network, each platform node is also responsible for receiving incoming data from other nodes, and relaying those data further upwards to the sink.

The platform sink serves as a collector of the monitored sensor data coming from the platform nodes. The sink itself does not connect to any sensor device, but uses BLE to connect to other peripherals in order to analyze and/or visualize the incoming data. As shown in Figure A.1, it sends the monitored data to a tablet which is connected to the proposed real-time feedback system.

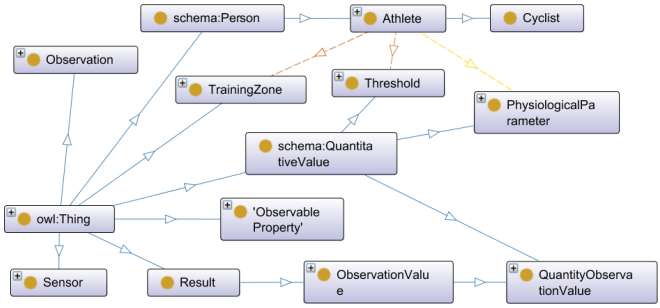


Figure A.4: Core structure of the cycling ontology[5]

A.5.2 Data dissemination

The IoT platform supports end-to-end dissemination of real-time data from sensors to the sink and in the reverse direction over a range up to several kilometers. Rather than using external infrastructure (e.g., 4G or LPWAN), it employs a multi-hop communication network using the IEEE 802.15.4g physical layer in combination with IEEE 802.15.4e MAC layer. The IEEE 802.15.4g amendment was recently added to the IEEE 802.15.4 standard, supports an infrastructure-less mode, and is tailored for low-power, long-range communication using the 868 MHz frequency band. Communication at this sub-1GHz frequency band allows for a connection between two nodes up to several hundreds of meters. The IEEE 802.15.4g physical layer is combined with the 6TiSCH architecture that combines industrial performance in terms of reliability and power consumption with a full IPv6-enabled IoT upper stack [26]. The key component of the 6TiSCH stack is the TSCH mode of the IEEE 802.15.4e MAC layer that combines channel hopping to avoid external interference and multi-path fading with a TDMA-based TX/RX schedule. This results in an extremely reliable and energy-efficient multi-hop IoT network, that can disseminate data in real-time over a distance of several kilometers. In the PoC set-up shown in Figure A.1, there was a direct 6TiSCH connection between the nodes and the sink.

A.6 Real-time feedback system

The main part of the feedback system is a stream reasoning framework running on a low-end device. To enable visual feedback to the user, a front-end application, such as the locally running web application shown in Figure A.3, should be implemented. Such an application can be viewed on the low-end device itself, or on any other device connecting to the same network.

[5]Created with the Protégé ontology editor (https://protege.stanford.edu).

Listing A.1: Example of a heart rate sensor observation, described in JSON-LD

```
{
  "@context": {
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "schema": "http://schema.org/",
    "sosa": "http://www.w3.org/ns/sosa/",
    "madeBySensor": { "@id": "sosa:madeBySensor", "@type": "@id" },
    "resultTime": { "@id": "sosa:resultTime", "@type": "xsd:dateTime" },
    "hasResult": { "@id": "sosa:hasResult" },
    "value": { "@id": "schema:value", "@type": "xsd:float" },
    "unitText": { "@id": "schema:unitText", "@type": "xsd:string" }
  },
  "@id": "http://idlab.ugent.be/conamo/sensors/
         Observation_HeartRateSensor_1593_2017_09_16_21_33_25_850Z",
  "@type": "http://www.w3.org/ns/sosa/HeartRateObservation",
  "madeBySensor": "http://idlab.ugent.be/conamo/sensors#HeartRateSensor_1593",
  "resultTime": "2017-09-16T21:33:25.850Z",
  "hasResult": {
    "@id": "http://idlab.ugent.be/conamo/sensors/
           ObservationValueSensor_HeartRateSensor_1593_2017_09_16_21_33_25_850Z",
    "@type": "http://idlab.ugent.be/conamo-vocab/sosa#HeartRateObservationValue",
    "value": "137", "unitText": "bmp"
  }
}
```

A.6.1 Ontology

A cycling ontology was developed[6], which is based on the SOSA ontology[7] (Sensor, Observation, Sample and Actuator). An overview of the core structure of the ontology is shown in Figure A.4. The ontology allows to define a cyclist, a training zone and the physiological profile of the cyclist, including training zone boundaries. Each training zone is annotated with comments containing feedback compiled by domain experts. Furthermore, sensor observations can be defined. Support is provided for location and quantity observation values. The latter include all relevant quantity observations with a value & unit, e.g., heart rate, altitude, power and speed of a cyclist. This generic definition enables to define corresponding generic reusable queries, as is explained in Section A.6.3.

The ontology is used to construct the knowledge base used by the RSP engine. Hence, queries and data (i.e., both static context data and input stream data), all need to be modeled in this ontology. Listing A.1 shows an example of a heart rate sensor observation as it is posted on the stream, described in JSON-LD.

A.6.2 Usage of C-SPARQL

The RSP engine used in the system is C-SPARQL, because of its supports for static context data, which CQELS has not. In C-SPARQL, a registered continuous query is executed when the corresponding window is triggered, i.e., after the slide.

[6] Publicly available at https://github.com/IBCNServices/cyclists-monitoring.
[7] http://www.w3.org/ns/sosa

Listing A.2: `getQuantityObservationValue` query

```
REGISTER STREAM getQuantityObservationValue AS

PREFIX f: <http://larkc.eu/csparql/sparql/jena/ext#>
PREFIX schema: <http://schema.org/>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX conamo-profile: <http://idlab.ugent.be/conamo-vocab/profile#>
PREFIX conamo-sosa: <http://idlab.ugent.be/conamo-vocab/sosa#>

SELECT ?deviceUUID ?firstName ?lastName ?gender ?birthDate ?sensor ?o ?time ?unit ?value
FROM STREAM <http://idlab.ugent.be/conamo/stream> [RANGE 10s STEP 1s]
FROM <http://localhost:8080/conamo/context/conamo-sensors.rdf>
FROM <http://localhost:8080/conamo/context/conamo-riders.rdf>
WHERE  {
    ?o sosa:hasResult ?ov . ?o sosa:resultTime ?time .
    ?sensor sosa:isHostedBy ?platform .
    ?platform conamo-sosa:UUID ?deviceUUID .
    ?rider conamo-profile:monitoredBy ?platform .
    ?rider schema:givenName ?firstName . ?rider schema:familyName ?lastName .
    ?rider schema:gender ?gender . ?rider schema:birthDate ?birthDate .
    ?ov a conamo-sosa:QuantityObservationValue .
    ?ov schema:unitText ?unit . ?ov schema:value ?value .
    { SELECT ?sensor ( MAX ( f:timestamp (?x, sosa:madeBySensor, ?sensor) ) AS ?ts )
      WHERE { ?x sosa:madeBySensor ?sensor . }
      GROUP BY ?sensor }
    FILTER ( f:timestamp (?o, sosa:madeBySensor, ?sensor) = ?ts )
}
```

To be able to work with a RESTful interface, the RSP Service Interface for C-SPARQL[8] is used. Upon start-up, a WebSocket server is started. For each registered query, the query output stream with the variable bindings is published on a query-specific WebSocket URL. Any interested client service, e.g., a web application, can then open a WebSocket connection to the relevant URL, where the variable bindings can then be received in real-time as messages.

A.6.3 C-SPARQL queries

To enable real-time feedback, C-SPARQL queries can be defined & registered in the engine. Two queries were constructed that allow feedback on a rider's heart rate and corresponding training zone.

The `getQuantityObservationValue` query (Listing A.2) analyzes the data to see if there is any `QuantityObservationValue`, e.g., the subclass `HeartRateObservationValue`, in the stream that can be linked to the rider's profile. Any observation of a sensor that produces a value with a unit as result, and is associated to an existing rider profile, is filtered by the query. For each sensor, only the most recent observation is outputted. The generic structure of the query exploits the generic structure of the cycling ontology explained in Section A.6.1, as it aggregates results of all sensors producing `QuantityObservationValue`s.

[8] https://github.com/streamreasoning/rsp-services-csparql

Listing A.3: `getTrainingZone` query

```
REGISTER STREAM getTrainingZone AS

PREFIX f: <http://larkc.eu/csparql/sparql/jena/ext#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX schema: <http://schema.org/>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX conamo-profile: <http://idlab.ugent.be/conamo-vocab/profile#>
PREFIX conamo-sosa: <http://idlab.ugent.be/conamo-vocab/sosa#>

SELECT
    ?deviceUUID ?firstName ?lastName ?gender ?birthDate ?sensor ?o ?time ?tzName
    ?tzDescription ?tzLB ?tzUB
FROM STREAM <http://idlab.ugent.be/conamo/stream> [RANGE 10s STEP 1s]
FROM <http://localhost:8080/conamo/context/conamo-sensors.rdf>
FROM <http://localhost:8080/conamo/context/conamo-riders.rdf>
WHERE  {
    ?o sosa:hasResult ?ov . ?o sosa:resultTime ?time .
    ?sensor sosa:isHostedBy ?platform .
    ?platform conamo-sosa:UUID ?deviceUUID .
    ?rider conamo-profile:monitoredBy ?platform .
    ?rider schema:givenName ?firstName . ?rider schema:familyName ?lastName .
    ?rider schema:gender ?gender. ?rider schema:birthDate ?birthDate .
    ?ov a conamo-sosa:HeartRateObservationValue .
    ?ov schema:value ?value .
    ?rider conamo-profile:hasTrainingZone ?tzRider .
    ?tzRider conamo-profile:hasLowerBound ?tzLB .
    ?tzRider conamo-profile:hasUpperBound ?tzUB .
    ?tzRider a ?tz . ?tz rdfs:label ?tzName .
    ?tz rdfs:comment ?tzDescription .
    ?tz rdfs:subClassOf conamo-profile:TrainingZone .
    { SELECT ?sensor ( MAX ( f:timestamp (?x, sosa:madeBySensor, ?sensor) ) AS ?ts )
      WHERE { ?x sosa:madeBySensor ?sensor . }
      GROUP BY ?sensor }
    FILTER ( f:timestamp (?o, sosa:madeBySensor, ?sensor) = ?ts )
    FILTER ( ?value >= ?tzLB ) FILTER ( ?value <= ?tzUB )
}
```

The `getTrainingZone` query (Listing A.3) derives the current heart rate zone of the riders. For any `HeartRateObservationValue` in the input window, the query tries to link this observation to the corresponding rider's profile, to select the heart rate training zone corresponding to the heart rate value. The associated training zone feedback is given in the result as well, as defined in the cycling ontology. For each heart rate sensor known by the system, only the most recent heart rate sensor observation is considered.

Both queries have the same window parameters. This is required for the real-time feedback to be consistent: if the heart rate value of a rider changes, his training zone changes accordingly.

A.6.4 Dynamic approach

The advantage of the described framework is its dynamic aspect. To create personalized feedback, only the context information needs to be updated while the

feedback system is running. For example, assume a new rider is joining the group of amateur cyclists. To be able to also output query results concerning this person's heart rate and training zone, the static context data used by C-SPARQL only needs to be updated by adding the sensor and profile information related to this rider and his bike. Besides, no runtime changes are needed to the C-SPARQL back-end server itself and the registered C-SPARQL queries. If a front-end application is able to deal with an arbitrary amount of riders, this application can simply continue running as before; the new bindings related to the new rider will also be sent as messages to the relevant WebSocket URLs. Similarly, new sensor types, new training zones or additional feedback can all be added to the context data, i.e., ontology, without having to change anything to the implemented platform. Also, for new queries, the query will be continuously executed once registered, sending its results as messages on its new WebSocket URL. Summarized, context data and queries can all be added, removed or updated at any time while running the system, without the need of other adaptations.

A.7 Evaluation set-up

It is important that the real-time feedback system, described in Section A.6, is capable of handling a group of amateur cyclists. However, more riders means more sensor observations and more static context data for a C-SPARQL query to work with. As low-end devices are typically characterized by limited memory and limited CPU capabilities, tests have been performed to assess how many riders can be supported by one Raspberry Pi. Furthermore, the impact of the heart rate frequency and C-SPARQL query parameters on the performance has been evaluated as well. The tests have been performed on the use case described in Section A.3, with the set-up described in Section A.4, and with the stream reasoning framework and queries as given in Section A.6.

A.7.1 Hardware specifications

The used low-end device for the tests described in this section is a Raspberry Pi 3, Model B. This device has a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM and MicroSD storage[9]. The operating system is Raspbian Stretch with desktop, version of November 2017, with kernel version 4.9[10]. The desktop version is chosen over the minimal OS, providing the ability to run and show a front-end application on the same device. This application may then consume the C-SPARQL query results to visualize the real-time feedback. When no foreground app is running on the Raspberry Pi, approximately 900MB of the memory is unused, and the active CPU usage is not higher than 1%.

[9] https://www.raspberrypi.org/products/raspberry-pi-3-model-b
[10] https://www.raspberrypi.org/downloads/raspbian

A.7.2 Evaluation parameters

First, not all available system memory may be in use when the feedback system is running. If that happens, memory issues may arise, disallowing the system to run as it should. Hence, the total memory usage should be lower than the available memory.

Second, the feedback should remain real-time. In concrete, this means that the system cannot start lagging, i.e., the total query execution time may not surpass the window sliding step. If that happens, the same query is triggered to execute again, while the previous execution is still ongoing.

Third, feedback should be updated real-time for each rider. This means that if the static context data describes n rider profiles, each query output should preferably contain n results, i.e., 1 for each rider[11], as it is guaranteed that only the most recent observation of each sensor is filtered by both queries. Hence, if this amount is smaller than n, no observation is present in the window for at least one rider. This is an indication that C-SPARQL is not able to run as it should, because it needs more resources than available.

A.7.3 Test scenarios

To perform the evaluation, a simulation of a real-life set-up has been performed. Different scenarios have been simulated. First, the *number of riders in the system* has been evaluated. With respect to the value of the *sliding step* of both C-SPARQL queries, the distinction between the ideal and extreme case has been made. In the ideal case, the queries are executed every second, to obtain an output event rate of approximately 1 second. This would then allow any front-end application to be updated every second, which is desirable for a real-time system. However, in the extreme case, one must consider the largest sliding step at which the updates can still be regarded as real-time. For heart rate values, this value is approximately 5 seconds, as a person's heart rate curve is typically relatively smooth. If the interval between updates is more than 5 seconds, this is no longer acceptable[12]. Besides the query sliding step, the impact of the *query window size* and the *event rate*, i.e., the time between successive incoming heart rate observations per rider, has also been investigated.

For each simulation, queries and context data were specified according to the query parameters and number of riders. Each time, a C-SPARQL[13] server has been started, and for each rider, a parallel script has then been launched to post randomly generated heart rate observations on the registered RDF stream at the specified event rate. In this way, the arrival pattern of heart rate observations of different sensors on the input stream is realistic: arrivals of the same period are close to each other in time, but their arrival order is undefined and can vary. To obtain comparable results,

[11] Assuming only active riders are described in the static data.
[12] According to the expertise of Energy Lab, partner of the CONAMO project.
[13] C-SPARQL version 0.9.7 (https://github.com/streamreasoning/CSPARQL-engine)

the amount of heart rate observations was chosen as such that both queries were executed at least 50 times during each simulation.

The measurements used for evaluation are taken from the C-SPARQL measurements provided in the generated log files. Based on the performance evaluation parameters defined in Section A.7.2, the following metrics were calculated for each simulation from the log output: (1) average query execution time[14], (2) 90th percentile of query execution times, (3) maximum query execution time, (4) average memory usage, (5) maximum memory usage, (6) normalized average number of query results. This last parameter is defined as the average number of output bindings per query execution, divided by the number of rider profiles. As at most one result per rider can be present, this number should always be 1 or smaller[15]. The closer this number is to 1, i.e., the closer the amount of results is to the number of riders, the better, as explained in Section A.7.2.

A.8 Results

In Figure A.5, the results of the evaluation of the number of riders in the system are shown, for a fixed event rate and query window size. In the plots, query execution times (average, 90th percentile, maximum) are displayed, as well as the normalized average number of query results. Memory usage is omitted from the plots, as these values never exceed 50MB. Results are compared between the ideal and the extreme case (query sliding step of 1 vs. 5 seconds). The plots show the simulation results for 1 to 20 riders.

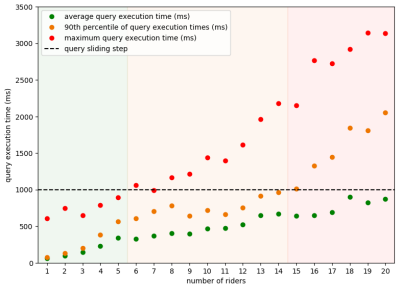
As observed in Figure A.5a and A.5b, the average query execution times never surpass the query sliding step. However, the maximum execution time does exceed this value, respectively from 6 riders and 13 riders onwards for the ideal and extreme case. The 90th percentile rises above this value from 15 and 19 riders on respectively. Based on these two events, the charts have been divided into three zones (green, orange, red), which are discussed in Section A.9. Associated with the rising execution times, the average number of query results decreases from 6 riders and onwards in the ideal case (Figure A.5c), with a significant drop for 15 to 20 riders. In the extreme case (Figure A.5d), this value only deviates from 1 for 15 riders and more.

The impact of the query window size on the calculated metrics is not plotted, as results show this impact is rather small. For a query sliding step of 1 second, 1 second between heart rate observations, and 5 rider profiles[16], the maximum query execution times are not impacted for a query window size increasing from 1 to 20. The average

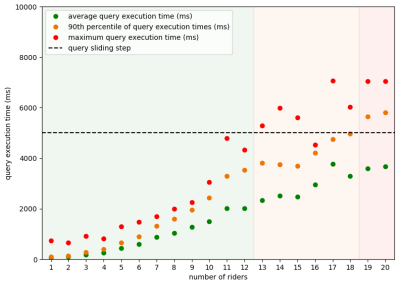
[14] As both queries are executed in parallel and on their own, the individual execution times are considered.

[15] When the load on C-SPARQL is too high, runtime exceptions can exceptionally lead to more results than normal. As these results cannot be used, these values are omitted when constructing the charts, to avoid misinterpretations.

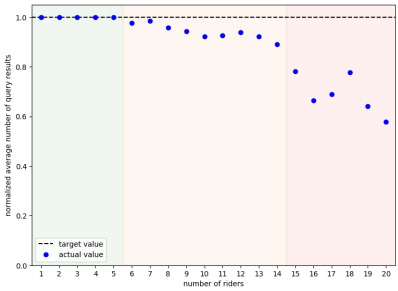
[16] The amount of riders was set to 5 because this is the highest number that is still in the green zone on the test results for a sliding step of 1 second in Figure A.5.



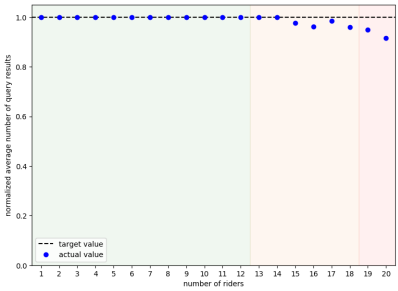
(a) Query execution times for a sliding step of 1 second (ideal case)



(b) Query execution times for a sliding step of 5 seconds (extreme case)



(c) Normalized average number of query results for a sliding step of 1 second (ideal case)



(d) Normalized average number of query results for a sliding step of 5 seconds (extreme case)

Figure A.5: Query execution times and normalized average number of query results for 1 to 20 rider profiles, measured on a Raspberry Pi 3, Model B. Time between incoming heart rates is 1 second for each rider. Query window size is 10 seconds.

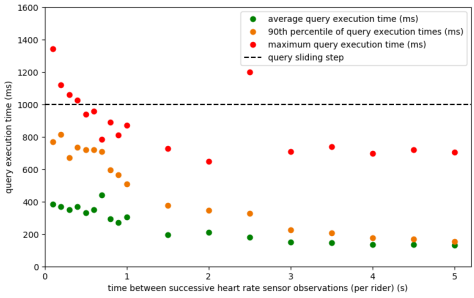


Figure A.6: Query execution times for varying time between successive heart rate observations per rider, for 5 riders. Query window size is 10 seconds, sliding step is 1 second.

increases from 95 to 324 seconds. The sliding step of 1 second is never exceeded by the 90th percentile, and not by more than 100ms by the maximum value. Memory usage does not exceed 20MB. Moreover, the number of results is not impacted at all.

Finally, the results of evaluating the impact of the event rate are shown in Figure A.6. With a fixed query window size of 10 seconds, sliding step of 1 second and 5 rider profiles[16], too high execution times only occur when the time between successive heart rate sensor observations gets 200ms and smaller. The 90th percentile however already starts increasing when this time gets smaller than 1 second (i.e., the sliding step), showing that high execution times start occurring more and more. Memory usage is not an issue, as it does not exceed 35MB. The amount of results is rot really impacted.

A.9 Discussion

The results of the performance evaluation described in Section A.8 show the bottlenecks of the Raspberry Pi as low-end device to run the stream reasoning framework of the real-time feedback system. Recalling the performance evaluation parameters discussed in Section A.7.2, it is clear that memory usage is not an issue for the use case described in Section A.3. The main performance bottlenecks are the query execution times exceeding the sliding step, causing performance issues resulting in fewer query results than expected.

As the green zone in the left of Figure A.5 shows, up to 5 riders can be supported without any problem in the ideal case of one query execution per second. In the orange zone, at least 90% of the query executions are finished within 1 second[17]. Issues occur but not yet constantly, and query executions do not always have results for each rider. Therefore, it is no longer ideal to use a Raspberry Pi, although the feedback may still be acceptably real-time at most moments. In the red zone however, high query execution times occur much more frequently, heavily impacting the amount of query results. As real-time feedback cannot be guaranteed any longer, it is not acceptable to use the system for this amount of riders.

Similarly, the same observations can be made in the extreme case of only one update every 5 seconds. However, compared to the ideal case, a group of 12 instead of 5 riders is still in the green zone, and 18 instead of 14 riders are still in the orange zone.

Figure A.6 shows that higher event rates lead to higher restrictions, especially when having more than one heart rate observation per rider per second. However, the rate of 1 observation per second, used for the evaluation in Figure A.5, is realistic for the set-up with the IoT platform described in Section A.5. Hence, it can be stated that one Raspberry Pi can support a group of 5 to 12 amateur cyclists to provide real-time

[17]Typically, in the orange zone, the maximum value occurs in the first query execution because of the materialization process, and other executions take much less time. This is no longer the case in the red zone.

feedback on their heart rate and training zones. This is acceptable in the given use case, as most groups do not consist of more than 12 people. Otherwise, it is always possible to parallelize the processing on more than one Raspberry Pi.

To perform a user evaluation, the use case scenario described in Section A.3 has been implemented for an end-to-end demo with two riders at the Media Fast Forward event[18] in December 2017. During the demo given, the users were generally interested in the real-time feedback given on the screen. Most user feedback received highlighted the added value of the personalization aspect. Some suggestive comments were made as well, mostly about the lack of integration of other sensors. Many people seemed to be interested in more extended feedback about other sensor data, i.e., not only limited to heart rate training zones. Also the integration of other context data such as route and weather information would be of interest when the system is applied for real-life cycling. This is easily possible by the use of semantics. In general, the generic ontology described in Section A.6.1 in combination with reasoning, allows to make any use case to be made context-aware and personalized to the desired extent.

A.10 Conclusion

In this appendix, a PoC of a personalized real-time feedback platform for amateur cyclists on low-end devices has been presented. By discussing a use case for heart rate and training zone feedback, it is explained how semantics and stream reasoning are applied to support the design of personalized and context-aware real-time services. In this way, amateur cyclists can monitor themselves, receive personalized feedback on how they are performing, and communicate with others during amateur cycling events. This is currently not yet possible with existing cycling training apps like Strava. A performance evaluation on a Raspberry Pi has shown that the system can give real-time heart rate and training zone feedback every 1 to 5 seconds for groups of up to 12 amateur cyclists. This demonstrates the potential of the framework to be used in real-life amateur cycling. Future work consists of adopting the system for extended use cases, addressing more than only the heart rate, e.g., power, location, speed, altitude, and taking into account more context information, e.g., the rider's power profile, the gradient at different route segments, weather information etc. This would illustrate the benefits of using semantics and reasoning even more.

[18] Organized annually in Brussels, Belgium by VRT (https://www.mediafastforward.be).

Funding

F. Ongenae is funded by BOF postdoc grant from UGent. Part of this research was funded by the FWO SBO S004017N IDEAL-IoT, the FWO SBO 150038 DiS-SeCt, and the imec ICON CONAMO. CONAMO is funded by imec, VLAIO, Rombit, Energy Lab and VRT.

Acknowledgments

Special thanks to Energy Lab for providing the domain expertise.

Availability of data and materials

Supportive information relative to this appendix is publicly available at https://github.com/IBCNServices/cyclists-monitoring/tree/master/poc-&-performance-evaluation. This repository contains details about the cycling ontology, relevant contextual data of the use case, queries used in the PoC presented in this appendix, and scripts to generate the simulation data used for the performance evaluation.

References

[1] P. O'Donoghue and L. Holmes. *Data analysis in sport*. Routledge, 2014.

[2] G. Romanillos, M. Zaltz Austwick, D. Ettema, and J. De Kruijf. *Big data and cycling*. Transport Reviews, 36(1):114–133, 2016. doi:10.1080/01441647.2015.1084067.

[3] R. Beecham and J. Wood. *Exploring gendered cycling behaviours within a large-scale behavioural data-set*. Transportation Planning and Technology, 37(1):83–97, 2014. doi:10.1080/03081060.2013.844903.

[4] G. Daneels, E. Municio, K. Spaey, G. Vandewiele, A. Dejonghe, F. Ongenae, S. Latré, and J. Famaey. *Real-time Data Dissemination and Analytics Platform for Challenging IoT Environments*. In 2017 Global Information Infrastructure and Networking Symposium (GIIS). IEEE, 2017. doi:10.1109/GIIS.2017.8169799.

[5] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. *Semantics for the Internet of Things: Early Progress and Back to the Future*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012. doi:10.4018/jswis.2012010101.

[6] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. *Stream reasoning: A survey and outlook*. Data Science, 1(1-2):59–83, 2017. doi:10.3233/DS-170006.

[7] A. Margara, J. Urbani, F. Van Harmelen, and H. Bal. *Streaming the web: Reasoning over dynamic data*. Journal of Web Semantics, 25:24–44, 2014. doi:10.1016/j.websem.2014.02.001.

[8] X. Su, E. Gilman, P. Wetz, J. Riekki, Y. Zuo, and T. Leppänen. *Stream reasoning for the Internet of Things: Challenges and gap analysis*. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), pages 1–10, New York, NY, USA, 2016. Association for Computing Machinery (ACM). doi:10.1145/2912845.2912853.

[9] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. *C-SPARQL: a continuous query language for RDF data streams*. International Journal of Semantic Computing, 4(1):3–25, 2010. doi:10.1142/S1793351X10000936.

[10] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. *A native and adaptive approach for unified processing of linked streams and linked data*. In The Semantic Web - ISWC 2011: Proceedings, Part I of the 10th International Semantic Web Conference, pages 370–388, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-25073-6_24.

[11] Y. A. Sedira, R. Tommasini, and E. Della Valle. *MobileWave: Publishing RDF Streams From SmartPhones*. In Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks, co-located with 16th International Semantic Web Conference (ISWC 2017). Springer, 2017. Available from: https://ceur-ws.org/Vol-1963/paper536.pdf.

[12] ITEA Office Association. *The Lifewear ITEA Project: Mobilized Lifestyle With Wearables*, 2013. Accessed: 2017-12-21. Available from: https://itea3.org/project/lifewear.html.

[13] G. Rubio Cifuentes, P. Castillejo Parrilla, J. F. Martínez Ortega, and E. Serral Asensio. *A novel context ontology to facilitate interoperation of semantic services in environments with wearable devices*. In OTM 2012: On the Move to Meaningful Internet Systems: OTM 2012 Workshops, pages 495–504. Springer, 2012. doi:10.1007/978-3-642-33618-8_66.

[14] J. Zhai and K. Zhou. *Semantic retrieval for sports information based on ontology and SPARQL*. In 2010 International Conference of Information Science and Management Engineering, pages 395–398. IEEE, 2010. doi:10.1109/ISME.2010.79.

[15] F. Camous, D. McCann, and M. Roantree. *Capturing personal health data from wearable sensors*. In 2008 International Symposium on Applications and the Internet, pages 153–156. IEEE, 2008. doi:10.1109/SAINT.2008.67.

[16] J. Lanza, L. Sánchez, V. Gutiérrez, J. A. Galache, J. R. Santana, P. Sotres, and L. Muñoz. *Smart City Services over a Future Internet Platform Based on Internet of Things and Cloud: The Smart Parking Case*. Energies, 9(9):719, 2016. doi:10.3390/en9090719.

[17] S. Latré, P. Leroux, T. Coenen, B. Braem, P. Ballon, and P. Demeester. *City of things: An integrated and multi-technology testbed for IoT smart city experiments*. In 2016 IEEE International Smart Cities Conference (ISC2), pages 1–8. IEEE, 2016. doi:10.1109/ISC2.2016.7580875.

[18] J. Jin, J. Gubbi, S. Marusic, and M. Palaniswami. *An Information Framework for Creating a Smart City Through Internet of Things*. IEEE Internet of Things Journal, 1(2):112–121, 2014. doi:10.1109/JIOT.2013.2296516.

[19] C. Pham, A. Rahim, and P. Cousin. *Low-cost, Long-range open IoT for smarter rural African villages*. In 2016 IEEE International Smart Cities Conference (ISC2), pages 1–6. IEEE, 2016. doi:10.1109/ISC2.2016.7580823.

[20] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi. *Long-range communications in unlicensed bands: The rising stars in the IoT and smart city scenarios*. IEEE Wireless Communications, 23(5):60–67, 2016. doi:10.1109/MWC.2016.7721743.

[21] S. Aust and T. Ito. *Sub 1GHz wireless LAN deployment scenarios and design implications in rural areas*. In 2011 IEEE GLOBECOM Workshops (GC Wkshps), pages 1045–1049. IEEE, 2011. doi:10.1109/GLOCOMW.2011.6162336.

[22] J. Gozalvez. *New 3GPP Standard for IoT [Mobile Radio]*. IEEE Vehicular Technology Magazine, 11(1):14–20, 2016. doi:10.1109/MVT.2015.2512358.

[23] I. Foche-Pérez, J. Simó-Reigadas, I. Prieto-Egido, E. Morgado, and A. Martínez-Fernández. *A dual IEEE 802.11 and IEEE 802.15–4 network architecture for energy-efficient communications with low-demanding applications*. Ad Hoc Networks, 37:337–353, 2016. doi:10.1016/j.adhoc.2015.08.028.

[24] Z. Zaidi and K.-c. Lan. *Wireless multihop backhauls for rural areas: A preliminary study*. PLoS ONE, 12(4):e0175358, 2017. doi:10.1371/journal.pone.0175358.

[25] M. Kent. *The Oxford Dictionary of Sports Science & Medicine*. Oxford University Press, 2006. doi:10.1093/acref/9780198568506.001.0001.

[26] X. Vilajosana, K. Pister, and T. Watteyne. *Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration*. RFC – Best Current Practice 8180, Internet Engineering Task Force (IETF), 2017. Available from: https://datatracker.ietf.org/doc/rfc8180/.

B

mBrain: Towards the Continuous Follow-up and Headache Classification of Primary Headache Disorder Patients

In Chapter 4, the generic cascading reasoning framework (as presented in Chapter 2) and the methodological design of the semantic IoT platform component DIVIDE (as presented in Chapter 3) were applied to the headache monitoring use case UC3. This use case is about the continuous monitoring of symptoms and triggers of patients diagnosed with a primary headache disorder. This use case is associated to the mBrain study, which is an observational study investigating the move to continuous, objective follow-up of headache patients. In this appendix, additional context about the mBrain study is provided to the interested reader. This appendix presents the full design of the data collection set-up of the mBrain study, and discusses the design of a knowledge-driven classification system for individual headache attacks. This system was already briefly presented in Chapter 4. It involves the usage of semantics, but it does not involve cascading reasoning or DIVIDE. Hence, this appendix does not specifically address any of the research challenges or contributions of this dissertation. It is thus purely present as additional context to the interested reader, and is not essential to understand the research of this doctoral dissertation.

⋆ ⋆ ⋆

M. De Brouwer[1], N. Vandenbussche[1], B. Steenwinckel, M. Stojchevska, J. Van Der Donckt, V. Degraeve, J. Vaneessen, F. De Turck, B. Volckaert, P. Boon, K. Paemeleire, S. Van Hoecke, and F. Ongenae

Published in BMC Medical Informatics and Decision Making Journal, Volume 22, March 2022.

Abstract

Background. The diagnosis of headache disorders relies on the correct classification of individual headache attacks. Currently, this is mainly done by clinicians in a clinical setting, which is dependent on subjective self-reported input from patients. Existing classification apps also rely on self-reported information and lack validation. Therefore, the exploratory mBrain study investigates moving to continuous, semi-autonomous and objective follow-up and classification based on both self-reported and objective physiological & contextual data.

Methods. The data collection set-up of the observational, longitudinal mBrain study involved physiological data from the Empatica E4 wearable, data-driven machine learning (ML) algorithms detecting activity, stress and sleep events from the wearables' data modalities, and a custom-made application to interact with these events and keep a diary of contextual and headache-specific data. A knowledge-based classification system for individual headache attacks was designed, focusing on migraine, cluster headache (CH) and tension-type headache (TTH) attacks, by using the classification criteria of ICHD-3. To show how headache and physiological data can be linked, a basic knowledge-based system for headache trigger detection is presented.

Results. In two waves, 14 migraine and 4 CH patients participated (mean duration 22.3 days). 133 headache attacks were registered (98 by migraine, 35 by CH patients). Strictly applying ICHD-3 criteria leads to 8/98 migraine without aura and 0/35 CH classifications. Adapted versions yield 28/98 migraine without aura and 17/35 CH classifications, with 12/18 participants having mostly diagnosis classifications when episodic TTH classifications (57/98 & 32/35) are ignored.

Conclusions. Strictly applying the ICHD-3 criteria on individual attacks does not yield good classification results. Adapted versions yield better results, with the mostly classified phenotype (migraine without aura vs. CH) matching the diagnosis for 12/18 patients. The absolute number of migraine without aura and CH classifications is, however, rather low. Example cases can be identified where activity and stress events explain patient-reported headache triggers. Continuous improvement of the data collection protocol, ML algorithms, and headache classification criteria (including the investigation of integrating physiological data), will further improve future headache follow-up, classification and trigger detection.

[1]The first two authors of this work contributed equally.

B.1 Background

B.1.1 Introduction

Headache disorders are highly prevalent conditions and among the most disabling disorders globally [1]. In 2016, it was estimated by the World Health Organization (WHO) that approximately 1 in 2 adults had experienced a headache disorder at least once in the last year [2].

According to the International Classification of Headache Disorders, Third Edition (ICHD-3), headaches can be characterized as either primary or secondary [3]. Primary headaches are those for which the headache and its associated features are the disorder itself, while secondary headaches are caused by an underlying disorder [4]. The most common primary headache disorders are migraine, cluster headache (CH) and tension-type headache (TTH). Migraine is characterized by disabling headache attacks that last 4 to 72 hours on average (if untreated) and are associated with symptoms, e.g., hypersensitivity to light and/or sound, nausea and/or vomiting. CH is characterized by shorter, severe and strictly unilateral headache attacks around the orbit or temple with associated symptoms such as restlessness and prominent ipsilateral cranial autonomic features. TTH, an almost universal experience for humans throughout life, was the third most prevalent condition worldwide in the Global Burden of Disease (GBD) study of 2016, with over one-fourth of the earth's population having the occasional attack of TTH per year [5]. In addition to eliciting pain, headache disorders also affect different physiological systems such as the autonomic nervous system and the homeostatic mechanisms, leading to symptoms such as fatigue, gastro-intestinal alterations and hypersensitivity to stimuli. These symptoms may even be present hours to days before the actual headache attacks and bring additional disability to the patient [6, 7]. The discovery of these different phases(i.e., prodromal, aura, headache, postdromal) during headache attacks, highlights the fact that headache disorders are neurological disorders that encompass dynamic neurophysiological alterations both before and after headache attacks and thus not only during painful periods [6, 8]. In the absence of biological markers to reliably diagnose the different primary headache disorders, physicians rely on accurate classification of headache attacks to correctly diagnose patients, and to continuously follow up with them to optimize their therapy and health management [9, 10]. In current clinical practice, both diagnosis and follow-up happen intermittently during a doctor's consultation through dialogue between patient and doctor. This means that physicians are dependent on intermittent subjective self-reporting by patients of historically experienced headache attacks and contextual factors (e.g., triggers).

The current gold standard for the diagnosis of headache disorders is the most recent ICHD-3, published by the International Headache Society (IHS) in 2018 [3]. The classification, established by headache experts, defines different diagnostic categories with specific criteria to classify a series of attacks as a certain disorder type, based

on the information collected through dialogue between patient and physician (e.g., the nosological description and duration of several individual headache attacks) [11]. An ICHD-3 diagnosis is made when the attacks described by the patient match the criteria for an ICHD-3 defined disorder and when there is no better explanation for the symptomatology. Despite this, ICHD-3 only contains criteria for disorders and not for separate headache attacks as such for continuous follow-up. They are not designed and have not been field tested to classify individual headache attacks. In fact, today, no generic evidence-based system exists that is able to autonomously classify the type of an individual headache attack [12].

To follow up on headache attacks, paper diaries and different apps such as Migraine Buddy [13] are being used. Some apps offer features which try to move towards a more continuous follow-up outside a clinical setting but are still mainly dependent on self-reported information [14]. Hence, the physiological aspect is currently not taken into account during headache follow-up. In general, up to now, little to no exploratory research has been done to measure the physiological impact of having a primary headache disorder on a person's lifestyle.

In the light of these opportunities and shortcomings associated with the current common practice, the exploratory mBrain study was started. Its main goal is to investigate how to move from the intermittent, subjective follow-up and classification of headache attacks and disorders based on self-reported data only, towards more continuous, prospective, semi-autonomous, multivariate and objective follow-up and classification, based on a combination of self-reported data and objective physiological and contextual data. mBrain is an observational and longitudinal study that focuses on patients diagnosed with episodic migraine with or without aura (ICHD-3 1.1 or 1.2), episodic cluster headache (ICHD-3 3.1.1), or chronic cluster headache (ICHD-3 3.1.2). During a trial period of approximately three weeks, participating patients are equipped with a wearable device and a smartphone that contains different applications used for data collection and follow-up. This way, the patient's physiological and contextual data is collected from two sources: automatically via the wearable device and built-in smartphone sensors, and via a mobile application with a diary of the patient's self-reported headache attacks. Using in-house designed machine learning (ML) algorithms, the automatically collected data is used to detect and recognize the patient's physical activities (e.g. sitting, walking etc.), sleeping periods and stress periods. These predictions are shown in the custom developed mobile application and can either be confirmed or corrected by the patient.

The main research question of the mBrain study is whether its approach allows to generate new insights that can have a positive impact on the continuous follow-up and diagnosis of primary headache disorders. This way, the high amount of collected information about the patient-centered ecological context could allow for a better understanding of the impact of primary headache disorders on patient lives and vice versa. If the answer to this question is positive, mBrain might be valuable to both

patients and doctors in several ways, for example by improving diagnosis through the help of an automatic headache classification system, the personalization of treatment plans, or the prediction of future headache attacks.

B.1.2 International Classification of Headache Disorders, 3rd edition

As a basis for the development of an autonomous headache classification system, the obvious starting point is the set of ICHD-3 criteria, since it is the current standard for diagnosing headache disorders. However, the fact that the ICHD-3 criteria are designed to be used by a physician during consultations, raises the question whether they can be applied as classification criteria in a continuous follow-up setting outside the walls of the physician's office, as is intended during the mBrain study. In addition, since they are designed to diagnose a disorder based on multiple headache attacks, it should be researched whether criteria for the classification of individual headache attacks can be extracted.

ICHD-3 already mentions the benefit of keeping a headache diary in which important characteristics of headache attacks are recorded. Apart from being helpful in teaching patients to differentiate between different headache types themselves, it has been shown that this improves the accuracy of clinical diagnoses [15, 16].

B.1.3 Related work

Today, different digital tools exist that can be used by headache patients to follow up on their headache syndromes [14]. Most of them are commercially available smartphone applications that solely focus on migraine [14]. The most well-known, most downloaded and highest rated application in this area is Migraine Buddy [13]. Key features are customizable attack recording and automatic sleep detection purely based on smartphone data. Besides sleep and weather, the app is based on only self-reported information. Other applications focusing on migraine include iMigraine [17], Migraine Headache Diary HeadApp [18] and Migraine Monitor [19]. Some applications focus on specific other primary headache disorders, such as My Cluster Headache [20] and Tension Headache [21]. Some general logging applications exist, such as Headache Log [22].

In research, headache apps have helped patients to control their acute medication usage. As overuse may lead to chronification of headaches, this may help the outcome of medication withdrawal in patients with medication-overuse headaches [23]. In recent years, apps are also being tested to provide behavioral therapy and telemedicine for headache disorders [24].

In general, studies have shown that self-reporting apps can be effective tools towards the improvement of self-managing headache disorders, and the mediation of

the interaction between headache patients and their doctors [25, 26]. However, the evidence base to show their effectiveness and clinical safety is not strong, and psychometrics are almost never taken into account [24, 26, 27]. Therefore, the inclusion of electronic devices such as wearables offers interesting additional options for the follow-up of headaches [24, 28]. Doing all of this, the privacy of patients should never be exposed, which is still an issue in many commercial apps [29]. Other criteria that are often forgotten about but are also important, include the usability of the app and personalization features [14].

In addition, digital tools exist that focus on the autonomous classification and diagnosis of headache attacks [30]. Importantly, previous studies recommend combining clinical interviews and a diagnostic diary for the follow-up and diagnosis of headache disorders, and to never purely rely on an autonomous system [31]. Nevertheless, there is still a need for tools that combine both aspects, to support clinicians in the classification and diagnosis [30]. Especially for the classification of *individual* headache attacks, no evidence-based research has sufficiently investigated this path [12]. One study specifically examined the classification of individual attacks as either migraine or TTH [12], by slightly adapting the ICHD-3 diagnostic criteria. However, the classification algorithm was not designed to discern any other headache disorder types.

The emerging potential of healthcare technology has been recognized by the IHS. Recently, the Clinical Trials Subcommittee published a position paper on the implementation of health technology assessments in clinical trials for medications or medical devices for the acute and preventive treatment of migraine. They recognize the importance of new technologies for the collection and analysis of evidence for the treatment of migraine and to facilitate health technology assessments that account for the distinctive nature of migraine and the heterogeneity of the affected population [32]. This statement was published after the start of this research project and start of this trial.

B.1.4 Objective & appendix organization

Several subquestions of the main research question need to be investigated:

(a) How can a system be set up that collects objective, explicit data about a patient's headache attacks and relevant context?

(b) How should a system be designed that autonomously classifies individual headache attacks? What criteria can be used for this classification, based on the available collected data? Can the ICHD-3 criteria be used for this purpose?

(c) Can physiological wearable data give an accurate view on contextual information such as the patient's activities, stress periods, and sleeping behavior?

(d) Is it actually useful to objectively map the context of headache attacks experienced by patients with migraine and CH? How can this physiological, contextual

and headache-related data be linked to be valuable for the continuous follow-up and/or classification of headaches?

The objective of this appendix is to investigate these individual research questions to assess the potential benefit of the continuous follow-up of headache patients using a combination of objective and self-reported data. First, the appendix describes all details of the mBrain data collection set-up. Spread over two data collection waves, a total of 18 migraine or CH patients have already participated in the study. By analyzing various statistics and the impact of changes made between both waves, research questions (a) and (c) are reviewed. Second, the appendix proposes the design of a preliminary, autonomous, knowledge-based classification system for individual headache attacks, starting from ICHD-3. It is evaluated on the available data from headache attack registrations by the study participants, to answer research question (b), and to further identify the requirements needed to improve its design. Finally, the appendix investigates research question (d), by analyzing whether and how contextual information of a patient can be used in a knowledge-based trigger detection system and early warning system for headache events.

B.2 Methods

To let migraine and CH patients participate in the mBrain study, a data collection protocol was designed and deployed. This section covers all aspects of this set-up: the wearable, the data-driven ML algorithms, the mobile applications, the data collection protocol, and the technical system architecture. Moreover, it discusses the design of a preliminary version of a knowledge-based headache classification system for individual headache attacks, and a possible methodology for the knowledge-based detection of predefined headache triggers.

B.2.1 Wearable: the Empatica E4

The wearable used in the mBrain study to collect the participant's physiological data, is the Empatica E4 [33]. This is a CE-certified medical-grade wearable device that offers physiological data acquisition in real-time. It has an internal memory that can store up to 36 hours of data, but also offers the option to send the data in real-time over a Bluetooth Low Energy (BLE) connection to a smartphone. The Empatica E4 consists of different physiological sensors:

- a photoplethysmogram (PPG) sensor, which measures blood volume pulse (BVP) (64 Hz frequency), from which heart rate (HR) and the inter beat interval of the heart rate (IBI) can be derived
- a 3-axis accelerometer (32 Hz frequency)

- an electrodermal activity (EDA) sensor, which measures the galvanic skin response (GSR) (4 Hz frequency)
- an infrared thermopile, which measures skin temperature (4 Hz frequency)

In the mBrain study, the BLE streaming mode of the Empatica E4 is used.

B.2.2 Data-driven machine learning algorithms

To get insight in the activities, sleeping behavior, and perceived stress of a participant, different data-driven ML algorithms were designed. Their goal is to accurately predict these events from the preprocessed objective physiological data collected by the Empatica E4 wearable. Four different algorithms were designed:

- Activity recognition: this algorithm determines whether a person is sitting, standing, lying down, walking, running, or cycling. It computes statistical features of the accelerometer signal only on a rolling window of 15 seconds with 50% overlap, which are then fed to a catboost (gradient boosted trees) model every 7.5 seconds. Smoothing is used to correct (obvious) mispredictions and arrive to minute-level predictions. In the final step, these 1-minute predictions are aggregated per 5-minute interval according to a fixed set of rules.
- Commute detection: this algorithm determines whether a person is commuting, based the collected location data and the output of the activity recognition algorithm. For every predicted activity, it calculates the movement speed based on the time and distance covered between the first and last location sample in the prediction interval. If the moving speed is higher than 25km/h, the predicted activity type is corrected to commuting.
- Sleep detection: this algorithm determines the person's time-to-bed, get-up time, sleep duration and preliminary sleep quality measures taking into account the number of wake-up periods between the time-to-bed and get-up timestamps. The algorithm analyzes the values of the activity index as described by Cole et al., averaged over windows of 5 minutes, that exceed predefined thresholds [34]. Aggregated windows above the threshold shorter than 1 hour, defined within a large period of rather low activity index values, are reported as wake-up periods. Larger aggregated windows above the threshold are used to determine the time-to-bed and get-up time. As a post-processing step, an anomaly detection algorithm analyzes for which sleeping periods the measured sleep quality is significantly lower than normal.
- Stress detection: this algorithm predicts the probability on a minute-wise granularity that a person is experiencing acute stress versus no stress. Using wearable data, it measures the physiological component of the stress-response, which is

characterized by sympathetic nervous system (SNS) activations [35]. Note that not-stress related events, such as exercise, can also influence SNS activations. The algorithm makes use of the E4's skin conductance and temperature signals and is trained on the publicly available WESAD dataset [36].

More details on the activity recognition, commute detection and sleep detection algorithms can be found in Stojchevska et al. [37].

B.2.3 Mobile applications

Three mobile applications were installed on the participant's smartphone. Two Android applications were designed for this study: mBrain and Empatica Streamer. The third app, OwnTracks [38], is an external application used for location tracking.

During the first data collection wave of the mBrain study, feedback of participants on the mBrain application was collected. This feedback, together with some observations made by the researchers, was used to improve the mBrain v1 baseline application of the first wave, to mBrain v2 used for the second wave.

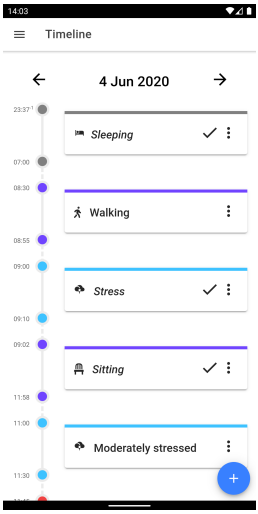
B.2.3.1 mBrain v1

The mBrain application is used by the participants to keep track of all relevant contextual data about their daily life. Screenshots of mBrain v1 are shown in Figure B.1.

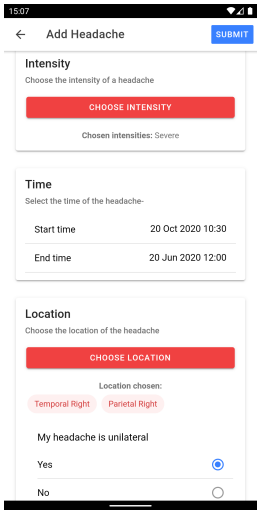
Account set-up Each participant of the mBrain study can set up an account in the mBrain app, in cooperation with the accompanying physician-researcher, and receives a unique patient ID. This patient ID is used to identify all collected data of this participant. Moreover, it is used by the accompanying physician-researcher to link the account to the concrete participating patient, which, for obvious privacy reasons, is unknown to the other non-medical researchers involved in the mBrain study. During set-up, the participant also selects his or her personal acute medications for headaches (e.g., analgesics such as acetaminophen or disorder-specific drugs such as triptans, ergotamines or oxygen therapy for CH).

Event registration The mBrain application allows the participant to keep track of different types of events: headache attacks, activities of daily living, sleeping periods, stress periods, medicine intakes, and (if applicable) menstrual periods. Table B.1 and Table B.2 detail the information that is requested for the registration of the different event types. In practice, participants register their headache attacks and medicine intakes themselves, and interact with activity, stress and sleep events that are automatically added to their timeline of events. Figure B.1b and B.1c show some screenshots of the registration of a headache attack.

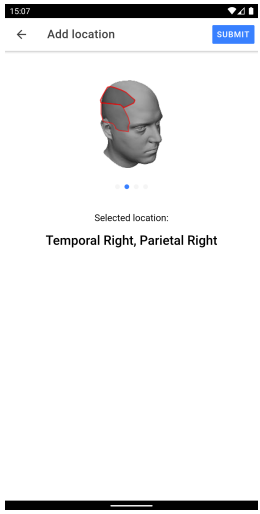
For the registration of headache attacks, an ICHD-3 based approach was developed by the physician-researchers of the mBrain study team to define the terminology



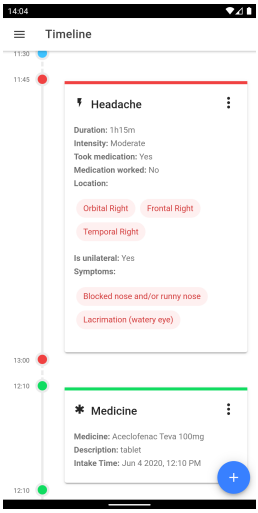
(a) Timeline with events



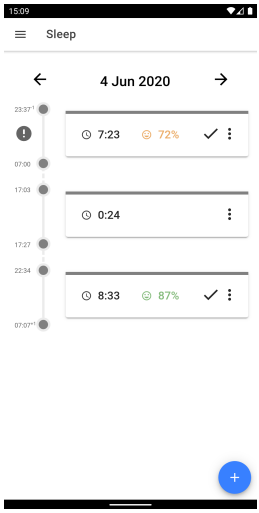
(b) Register headache attack



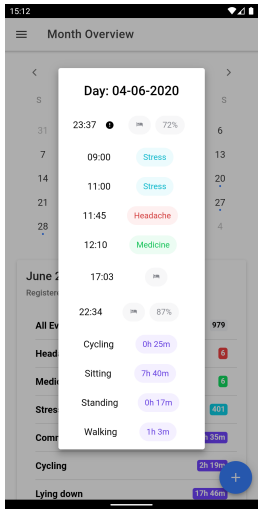
(c) Select headache location



(d) Timeline with new headache attack



(e) Sleep overview



(f) Month overview with daily summaries

Figure B.1: Screenshots of the mBrain v1 application

Table B.1: Information requested in the mBrain app for the registration of the different event types. For items with a predefined set of options, the options are mentioned between brackets, or the table with the options is referred to between brackets. In mBrain v1, all information except information with an asterisk (*) is required. In mBrain v2, all information is required.

Event type	Requested information
Stress	start time; end time; stress intensity (no stress (0), moderate stress (1), high stress (2))
Activity	start time; end time; activity type (sedentary, sitting, standing, lying down, walking, running, cycling, commuting, other [any type of activity is allowed, of which a textual description required])
Sleep	time to bed; wake-up time
Medicine intake	time; medicine name, dose & form
Headache attack	start time; end time; pain intensity (Table B.2); headache location(s) (Table B.2); pain being unilateral (yes, no); headache symptom(s)* (Table B.2); headache trigger(s)* (Table B.2); acute medication intake (yes and successful, yes but unsuccessful, no)
Period (if applicable)	start time; end time

Table B.2: Requested input and available options when registering a headache attack in the mBrain app. The information is applicable to both mBrain v1 & v2. For all information except pain intensity, more than one option can be selected by the participant. In mBrain v2, the option "none of those" can also be selected for headache symptoms and headache triggers if no other option is selected. The available options for pain intensity and headache symptoms are based on the diagnostic criteria of migraine and cluster headache in ICHD-3 [3].

Requested information	Available options
Pain intensity	no pain (0), mild pain (1), moderate pain (2), severe pain (3), very severe pain (4)
Headache location(s)	cervical left; cervical mid; cervical right; frontal left; frontal mid; frontal right; mandibular left; mandibular right; maxillar left; maxillar right; occipital left; occipital mid; occipital right; orbital left; orbital right; parietal left; parietal mid; parietal right; temporal left; temporal right
Headache symptom(s)	conjunctival injection; lacrimation; ptosis; miosis; eyelid oedema; nasal congestion; rhinorrhoea; sweaty forehead and face; pulsating pain; movement sensitivity / pain increment during routine physical activity; restlessness or agitation; photophobia; phonophobia; osmophobia; nausea; vomiting
Headache trigger(s)	alcohol; atmospheric pressure difference; bright light; caffeine; change in weather; cold; coughing; decreased water intake; flickering light; heat; height; holiday; illnesses; loud sounds; medication; menstrual cycle; physical exercise; pressing; relieve from stress; resolvents; sexual intercourse; skipping of meals; sleep deprivation; sleeping away; smells/odors; sneezing; specific head movements; stress; touch

used for pain severity, pain location and headache symptomatology. For severity, a five-point Likert scale approach was used in accordance with intensity levels of ICHD-3 (no pain, mild pain, moderate pain, severe pain and very-severe pain). For headache location, an interactive manikin with topographical anatomical landmarks was developed, where participants can register one or many zones of pain during a headache attack. The ICHD-3 terminology on headache associated symptoms was translated into layman's terms in Dutch by the physician-researchers in accordance with common clinical terminology in headache medicine, since no formal ICHD-3 translation in Dutch existed at the time of the study.

Timeline of events The timeline shows a chronological overview of all events in the selected day. It consists of events registered by the participant, and events automatically added by the ML algorithms. Overlapping events are show sequentially. Figure B.1a and B.1d show some screenshots of the timeline.

For the events that are automatically added by the data-driven ML algorithms, the participant is asked to interact with them *as much as possible* to let the system know whether these events are correct or not. In this way, the correctness of the ML algorithms can be validated, which is important for further improvement and allows to shift towards personalization of the algorithms. If an event is correct, the participant can easily confirm the event by hitting a check mark button. If an event is incorrect, the participant can edit or remove it.

Sleep overview Specifically for sleep events, the participant is able to view a dedicated sleep overview per day. Additional information is shown for automatically added sleep events, based on the output of the sleep ML algorithm: an estimation of the quality of the sleep (percentage) and a visual indication when this quality is significantly lower than normal (via an exclamation mark). An example of this sleep overview is visualized in Figure B.1e.

Daily records The participant is requested to fill in and submit daily reports with the following information: the general stress level during the day (on a scale from 1 to 10, 10 meaning the highest possible stress level); the general mood during the day (on a scale from 1 to 5, 5 meaning the best possible mood); whether or not each of the three main meals (breakfast, lunch, dinner) has been taken, and the time of consumption for taken meals. The participant can fill this in at any time.

Other functionality The participant can hit the event mark button on a connected Empatica E4 to register specific moments in time. These timestamps are saved and shown on the tag page for later use, e.g., to easily register a headache attack or edit an event predicted by the ML algorithms. Moreover, the month overview shows statistics about all events in the timeline, aggregated per month as well as summarized per day. A screenshot of this is shown in Figure B.1f.

B.2.3.2 mBrain v2

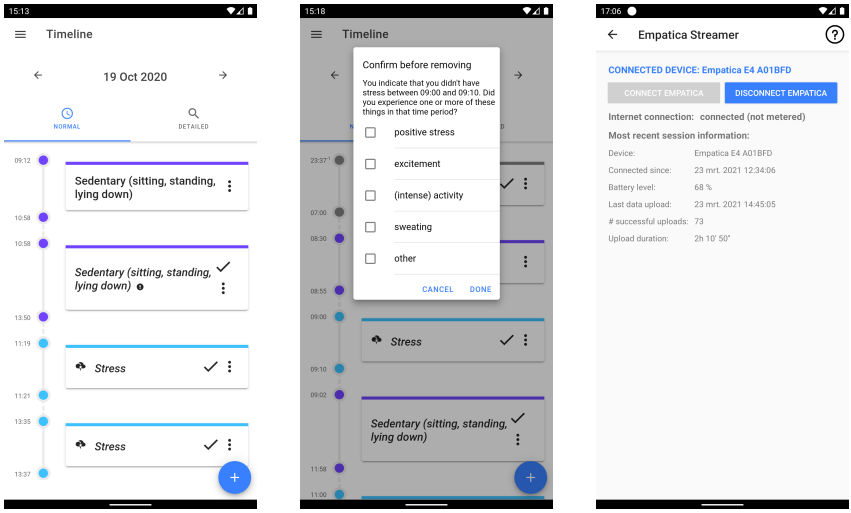
Based on observations by the researchers and feedback collected from participants of the first data collection wave with mBrain v1, mBrain v2 was created for usage during the second wave. This section highlights the changes of mBrain v2 compared to mBrain v1. A motivation for these changes and a discussion of their impact, is given in Section B.4.

Symptom and trigger input required for headache registration In mBrain v2, the requirement is added that a participant needs to select at least one symptom and trigger when registering a headache attack. To accommodate the situation where no symptom or trigger is applicable, the new option "none of those" can be selected.

Timeline changes The following things have changed in the timeline of mBrain v2:

- The timeline is split up in two views: a normal view and a detailed view. The detailed view shows the exact same timeline as in mBrain v1. The normal view is the new view that abstracts sedentary activities. A sedentary activity is an activity with type "sedentary", "sitting", "standing" or "lying down". All sedentary activities are labeled as "sedentary", and grouped when they follow up on each other in time (i.e., when there are at most 60 seconds in between), and when they are either all confirmed or all unconfirmed. This way, the number of individual events in the timeline is significantly reduced, since sedentary events take up the largest part of the detailed timeline. Interactions with sedentary events in the normal timeline view are reflected in the other timeline view. A confirmation of a sedentary event in the normal view only means that the event is confirmed as being of sedentary form, but not explicitly as *sitting*, *standing* or *lying down* if that is the predicted label, unless the individual event is also confirmed in the detailed timeline view. As especially dynamic behavior impacts migraine [39], this grouping of sedentary events has no negative impact on migraine management, but highly reduces the required feedback of participants: it allows the participant to give coarse-grained feedback whenever giving fine-grained feedback is not feasible. The normal view is the default view when opening the timeline page.

- The number of stress events that are automatically added to the participant's timeline by the stress detection ML algorithm is limited to at most 2 events per hour and at most 10 events per day. Within a single execution of the stress detection algorithm, the newly predicted stress periods are sorted from longest to shortest duration. Next, this list is processed in order, where each event is only added to the timeline if the applicable hourly and daily limits both have not yet been reached.



(a) mBrain v2: normal and detailed timeline view

(b) mBrain v2: remove predicted stress event

(c) Empatica Streamer: connected with an Empatica E4

Figure B.2: Screenshots of the mBrain v2 and Empatica Streamer applications

- Whenever a stress event is added to the timeline that ended not longer than 15 minutes ago, a mobile notification is sent to the smartphone of the participant.

- For every automatically added sedentary activity in the timeline, the participant's location at the start time and end time of the activity is compared. If there is a significant difference between both locations, the activity is flagged and a visual exclamation sign is added to this event in the timeline, indicating a potential misprediction to the participant.

In Figure B.2a, a screenshot of the updated timeline of mBrain v2 is shown.

Additional input requested when confirming or deleting a predicted stress event When the participant removes an incorrect automatically added stress event from the timeline in mBrain v2, he or she is asked to specify what was experienced during the time of the mispredicted period. At least one of the following options should be selected: positive stress; excitement; (intense) movement or activity; sweating; other (none of the available options). A screenshot of this new input request is shown in Figure B.2b. Moreover, the participant is explicitly requested to select the intensity (moderate stress or high stress) when confirming an automatically added stress event, as no stress level is predicted by the ML algorithm.

B.2.3.3 Empatica Streamer

The Empatica Streamer application is a separate application installed on the participant's smartphone, which can only be opened from the menu of the mBrain app. It allows the participant to connect an Empatica E4 device to the smartphone via BLE. Once connected, the Empatica E4 will stream the physiological data in real-time to the smartphone, which buffers the data and chronologically uploads it over WiFi to the server environment. Via the application and permanent notifications, the participant can keep track of the connection status. Figure B.2c shows a screenshot of the main page of the Empatica Streamer app while it has an active BLE connection with an Empatica E4 device.

B.2.3.4 OwnTracks

OwnTracks [38] is an external application that is used to collect the participant's location data. For the mBrain study, OwnTracks is configured to upload the smartphone's GPS coordinates and an estimated accuracy every X minutes, provided that there is at least a 50 meter difference to the previously uploaded coordinates. During the first data collection wave, X was set to 3 minutes. For the second wave, X was updated to 0.5 minutes, to allow for better route reconstruction for high velocity movements such as driving or cycling.

B.2.4 Protocol of data collection trial

The data collection trial with actual headache patients was performed in cooperation with physician-researchers from the Department of Neurology of the Ghent University Hospital. The protocol has been approved by the Ethics Committee of the Ghent University Hospital (BC-07403). The methods in the protocol are in accordance with all relevant guidelines and regulations.

B.2.4.1 Inclusion & exclusion criteria

To participate in the mBrain study, the inclusion criteria were defined as follows:

1. The patient is an adult between 18 and 65 years old.
2. Only one of the following two criteria is fulfilled:
   (a) The patient is diagnosed with *migraine with aura* or *migraine without aura*. The diagnosis is made based on the diagnostic criteria 1.1 or 1.2 of ICHD-3 [3], respectively. The diagnosis exists for at least 1 year.
   (b) The patient is diagnosed with *cluster headache*. The diagnosis is made based on the diagnostic criteria 3.1 of ICHD-3 [3]. The diagnosis exists for at least 1 year.

The following exclusion criteria were defined:

1. The patient is diagnosed with any other headache disorder (other than the ones specified by inclusion criterion 2) that makes the classification of a headache attack more difficult, with the exception of comorbid infrequent or frequent episodic TTH (ICHD-3 2.1 or 2.2) if those episodes are clearly distinguishable from attacks of migraine or CH [40].

2. The patient is suffering or has recently suffered from alcohol and/or drug abuse.

3. The patient has significant medical comorbidity that can cause interference with the research, according to the judgment of the physician-researcher.

4. The patient is traveling to a foreign country during the trial period.

5. The patient is using beta blockers.

6. The patient is participating in any other academic or commercial clinical trial.

Additional criteria for migraine patients For patients fulfilling inclusion criterion 2a, the following additional inclusion criteria were defined:

1. The patient has had less than 15 days with headache per month during the past 3 months.

2. The patient has at least 2 migraine attacks per month.

3. Migraine attacks started when the patient was younger than 50 years.

4. The patient's migraine attacks can be clearly distinguished from any other headache disorder.

Additional criteria for CH patients For patients fulfilling inclusion criterion 2b, the following additional inclusion criteria were defined:

1. CH attacks started when the patient was younger than 50 years.

2. The patient expects to have at least 5 CH attacks per week.

3. The patient's CH can be clearly distinguished from any other headache disorder.

Only patients with an Android smartphone could participate in the study. However, different Android smartphones were available for patients without an Android smartphone such that they could still participate, if all other criteria were fulfilled.

B.2.4.2 Patient recruitment & start-of-trial intake visit

Patients eligible to participate were recruited via the headache outpatient clinic and communication channels of the Department of Neurology of the Ghent University Hospital. The intake and outtake visits took place at Ghent University Hospital.

During the intake visit, the participant received detailed information from the physician-researcher about the goals of the study, the rights of the participants, and the study safety procedures. The participant was requested to read the information letter of the study and sign an Informed Consent Agreement. Next, the physician-researcher took a detailed history on baseline demographic characteristics, headache-related current and previous medication usage and headache characteristics. Thereafter, the physician asked the participant to fill in multiple questionnaires:

- The Migraine Disability Assessment Test (MIDAS), Dutch version [41–43]
- The MOS Short-Form General Health Survey (SF-20) [44, 45]
- Migraine-Specific Quality-of-Life Questionnaire (MSQ Version 2.1) [46] (only for patients diagnosed with *migraine with aura* or *migraine without aura*)

After this, the Empatica E4 was given to the participant. The three mobile applications were installed on the participant's smartphone. If the participant did not possess an Android smartphone, he or she was given a smartphone with the applications preinstalled. The physician-researcher set up the different applications together with the participant. Finally, the participant received a user manual with detailed instructions and guidelines about the trial and the different mobile applications. The most important instructions were also explained and demonstrated by the physician-researcher. Subjects did not receive any compensation for participating in the study apart from a parking ticket voucher.

B.2.4.3 Continuous follow-up during trial period

The trial period of each participant took 21 days, with a possible deviation of a few days depending on the participant's personal agenda. During this period, the participant was requested to adhere to the following guidelines:

- The operating system specific and application-specific settings of the installed applications must not be changed, except for the in-app settings of the mBrain application. Bluetooth should be enabled at all times to retain connection with the Empatica E4 and WiFi should be enabled as much as possible to enable the uploading of the collected data.
- The Empatica E4 wearable should be worn as much as possible. During these periods, the Empatica E4 should be connected to the smartphone via the

Empatica Streamer application as much as possible. During an active connection, the Empatica E4 and the smartphone used for the trial should be kept as close as possible to each other, to avoid a Bluetooth disconnection which causes the Empatica data streaming to stop until a manual reconnection. To avoid unintentional disconnections, the connection status should be regularly checked via the Empatica Streamer application and the Empatica device itself. Since the battery of the Empatica E4 lasts approximately 6 to 12 hours in streaming mode, the Empatica must be charged at least two times per day, but not during nighttime whilst sleeping. Instead, the preferred schedule is to charge the Empatica in the evening, wear a fully charged Empatica at least one hour before going to bed, and keep wearing the Empatica while sleeping.

- A correct and detailed diary of all events should be kept via the mBrain application. This request is threefold. First, automatically added events (activities, sleep periods and stress periods) in the event timeline should be interacted with as much as possible by either confirming, editing or removing them. Second, all other relevant events that are not automatically added to the timeline should be registered. This involves headache attacks, acute medicine intakes and menstrual periods (if applicable), as well as activities, sleep periods and stress periods that are not automatically added to the timeline. Third, the daily record should be filled in every day. For all information logged in the mBrain app, and especially for the registration of headache attacks, it is important to be as precise and complete as possible.

These guidelines were communicated to the participant during the trial intake visit and were also listed in the user manual given to the participant.

B.2.4.4 End-of-trial outtake visit

When the participant's trial period ended, he or she had a final outtake visit with the physician-researcher. During this visit, all used devices were returned. In preparation of the outtake visit, the participant's collected data was observed and analyzed by mBrain's researchers. Based on this, a report was created that the physician-researcher discussed with the participant during the outtake visit. This report contained questions to help resolving any missing or incomplete data observed during the analysis, questions to clarify interesting observations made by the researchers, and questions related to the interaction with and general feeling about the mobile applications. This process allowed to get new insights and improve further iterations of the applications and ML algorithms.

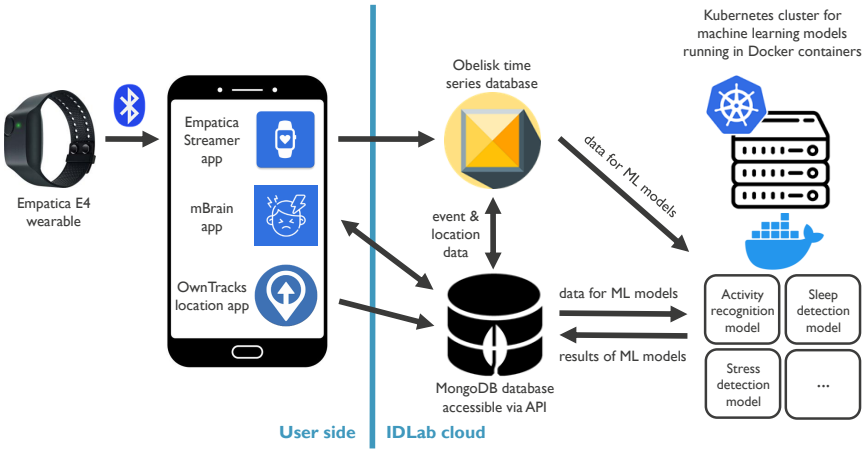


Figure B.3: Architectural set-up of the mBrain data collection system

B.2.5 Technical architecture

The architecture of the mBrain data collection system is shown in Figure B.3. It contains the user side and the server environment hosted on the IDLab cloud.

On the user side, the three mobile applications (mBrain, Empatica Streamer and OwnTracks) are installed on an Android smartphone. The Empatica Streamer app makes a connection with the Empatica E4, which streams its measured physiological data over a BLE connection to the smartphone. The applications communicate with the server environment over a secured HTTPS connection.

The server environment is hosted on the IDLab cloud, which is the research group's in-house cloud environment. For mBrain, it hosts three main parts. First, it hosts Obelisk, which is an existing platform for building scalable applications using time-series data [47]. Obelisk is used for storing all collected high-frequency Empatica data, except for the output of the PPG sensor, which was not stored due to technical constraints. This data is sent to Obelisk from the Empatica Streamer app for ingestion, and is available for querying to the other server components. Second, the server environment contains a MongoDB [48] database instance which stores all other mBrain-related data. It is accessible via the mBrain application programming interface (API). This API is used by the mBrain app for communicating to the server environment, and by OwnTracks to upload the patient's location data. Third, a Kubernetes [49] cluster is deployed on the iLab.t Virtual Wall [50] portion of the server environment. This cluster is used to automatically and reliably schedule and execute individual runs of the different ML algorithms in Docker containers [51]. The activity recognition, commute detection and stress detection algorithms are executed every

five minutes, while the sleep detection algorithm runs every 24 hours. These algorithms each query the relevant Empatica data from Obelisk for the processed time period, and fetch other relevant data from the MongoDB database via the mBrain API. Using this data, the algorithms run their predictive models and send the automatically generated events to the mBrain API.

B.2.6 Knowledge-based classification of individual headache attacks

This section proposes the design of a preliminary, autonomous, i.e., system-based, knowledge-based classification system for individual headache attacks, based on the data collected in the mBrain study, starting from the ICHD-3 diagnostic criteria.

B.2.6.1 Usage of semantics & design of mBrain ontology

For the design of the classification system, a semantic approach is applied. This enables the consolidation of the available data, as it imposes a common, machine-interpretable data representation [52]. To do so, the mBrain ontology has been designed. This ontology is a semantic model that formally describes the mBrain domain knowledge through its concepts and their relationships and attributes [53]. It allows to semantically describe all details of a headache attack and a patient's contextual information, to be used for the knowledge-based classification of headache attacks and detection of headache triggers.

For the headache-specific aspects, the mBrain ontology contains the classification hierarchy of the different headache disorders. For the primary headache disorders, the hierarchy is worked out more completely in terms of subcategories specified in ICHD-3. A semantic classification system needs to start from classifying an individual headache attack and could then potentially use these individual classifications to assess a person's general disorder diagnosis. Hence, the ontology makes a distinction between an individual headache attack that can be of a certain headache phenotype, and the general diagnosis of a patient. Moreover, the ontology should at least contain all concepts related to the ICHD-3 diagnostic criteria of the primary headache disorders focused on in the mBrain project, i.e., migraine, CH, and frequent/infrequent episodic TTH. Nevertheless, its generic design allows to easily extend the ontology for other headache disorder types and phenotypes in the future.

The mBrain ontology is built as an extension of the DAHCC (Data Analytics for Health and Connected Care) ontology [54]. This general ontology is internally designed to describe everything related to the collection of raw sensor data, contextual data and ML predictions. This linking is important to enable hybrid AI, where the data-driven ML outputs can be used in knowledge-based systems. The DAHCC ontology is a combination of multiple, existing ontologies, enriched with information and concepts specific for its purpose. These ontologies include SAREF4EHAW (the

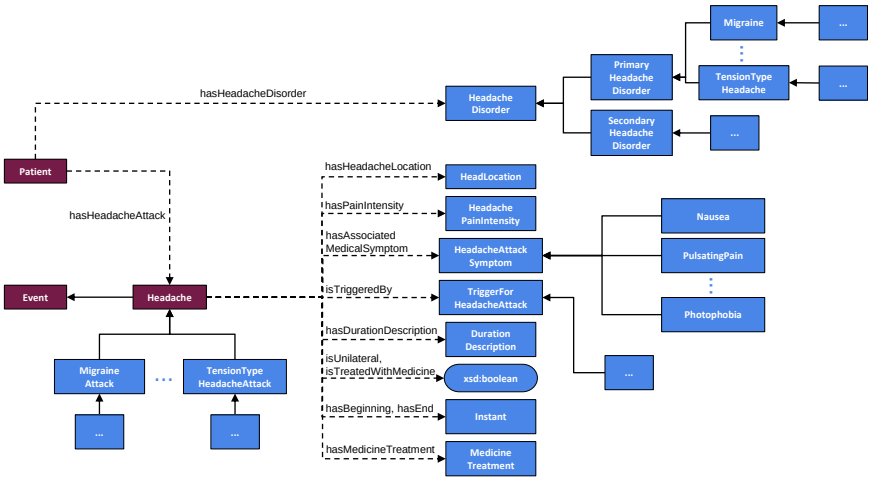


Figure B.4: Overview of the most important concepts in the mBrain ontology, and the relations between them. The blue concepts are the new concepts introduced in the mBrain ontology, while the red concepts are the concepts that exist in the DAHCC ontology, with which the mBrain ontology links.

Smart Applications REFerence ontology extended with concepts of the eHealth Ageing Well domain) [55] and SSN (Semantic Sensor Network) [56].

Figure B.4 shows a high-level overview of the most important concepts in the mBrain ontology, the relations between them, and the link with the DAHCC ontology. In Listing B.1, a semantic representation is given of a headache attack, predicted activity and predicted stress event with the mBrain ontology.

B.2.6.2 Requirements of the classification system

The inclusion criteria of the mBrain study allow patients diagnosed with *migraine without aura*, *migraine with aura*, and *CH* to participate in the study. Moreover, since TTH is the most common primary headache disorder [2, 3], it cannot be ignored. Hence, this classification system will focus on migraine, CH and TTH.

For the design of a knowledge-based classification system, ICHD-3 is chosen as the starting point. The focus of our research is on the headache attacks. We included patients with migraine with aura, however, we did not specifically investigate aura as a separate phenomenon. Therefore, in the headache classification system, the focus will be on migraine following the criteria of migraine without aura (ICHD-3 1.1). For TTH phenotype, we take note that ICHD-3 makes a further distinction between *infrequent* and *frequent* episodic TTH. However, the diagnostic criteria of both disorders only differ in the frequency of individual episodes, while the diagnostic criteria applicable to individual episodes are identical. Therefore, the classification for individual

Listing B.1: Semantic representation using the mBrain ontology of a registered headache attack, a predicted activity, and a predicted stress event, in RDF format

```
# mBrain ontology
@prefix medical: <http://contextaware.ilabt.imec.be/ontology/medical.owl#> .
@prefix headache: <http://contextaware.ilabt.imec.be/ontology/headache.owl#> .
# DAHCC ontology
@prefix DAHCC: <http://IBCNServices.github.io/DAHCC/>
@prefix saref4ehaw_ML: <http://IBCNServices.github.io/saref4ehaw_ML/>
# existing ontologies
@prefix DUL: <http://IBCNServices.github.io/Accio-Ontology/ontologies/DUL.owl#> .
@prefix saref4ehaw: <https://saref.etsi.org/saref4ehaw/>
@prefix time: <http://www.w3.org/2006/time#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# headache attack
DAHCC:5e3c2fcbe41ccb080597fc02 headache:hasHeadacheAttack DAHCC:5f02e8897235087baf826557 .
DAHCC:5f02e8897235087baf826557 rdf:type headache:HeadacheAttack ;
    medical:hasID "5f02e8897235087baf826557"^^xsd:string;
    time:hasBeginning
        [ rdf:type time:Instant ;
          time:inXSDDateTime "2020-07-06T07:00:00"^^xsd:dateTime ] ;
    time:hasEnd
        [ rdf:type time:Instant ;
          time:inXSDDateTime "2020-07-06T11:00:00"^^xsd:dateTime ] ;
    time:hasDurationDescription
        [ rdf:type time:DurationDescription ;
          time:hours "4.000000"^^xsd:decimal ] ;
    medical:hasPainIntensity [ rdf:type headache:Mild ] ;
    headache:hasHeadacheLocation
        [ headache:hasHeadRegion headache:FrontalHeadRegion ;
          headache:hasHeadSide headache:MidSideOfHead ] ;
    headache:isUnilateral "false"^^xsd:boolean ;
    medical:hasAssociatedMedicalSymptom [ rdf:type headache:PulsatingPain ] ;
    medical:hasAssociatedMedicalSymptom [ rdf:type medical:MovementSensitivity ] ;
    medical:hasAssociatedMedicalSymptom
        [ rdf:type headache:PainAggravationByRoutinePhysicalActivity ] ;
    medical:hasAssociatedMedicalSymptom [ rdf:type medical:Phonophobia ] ;
    medical:hasAssociatedMedicalSymptom [ rdf:type medical:Photophobia ] ;
    medical:isTriggeredBy [ rdf:type medical:Alcohol ] ;
    medical:isTriggeredBy [ rdf:type medical:SleepDeprivation ] ;
    medical:isTreatedWithMedicine "true"^^xsd:boolean ;
    medical:hasMedicineTreatment [ medical:isSuccessful "true"^^xsd:boolean ] .

# activity
DAHCC:5e3c2fcbe41ccb080597fc02 saref4ehaw:hasActivity
    DAHCC:5ff1fcccdf9ce11f35e37fe8 .
DAHCC:5ff1fcccdf9ce11f35e37fe8 rdf:type saref4ehaw:Activity ;
    rdf:type DAHCC:Sitting ;
    saref4ehaw_ML:isPredicted "true"^^xsd:boolean ;
    saref4ehaw:activityDuration "2040"^^xsd:float .

# stress event
DAHCC:5e3c2fcbe41ccb080597fc02 DAHCC:hasStressEvent
    DAHCC:5fef519c9db6713d42d4627a .
DAHCC:5fef519c9db6713d42d4627a rdf:type DAHCC:Stress ;
    saref4ehaw_ML:isPredicted "false"^^xsd:boolean ;
    DAHCC:isConfirmed "true"^^xsd:boolean ;
    DAHCC:stressLevel "1"^^xsd:float ;
    DAHCC:eventDuration "60"^^xsd:float .
```

headache episodes only differentiates between TTH or not. Note that the term infrequent or frequent does hence not matter for the type of the individual episode and will therefore be omitted for the remainder of this appendix. Other attack phenotypes such as thunderclap headache or stabbing/paroxysmal headaches, are currently out of the scope of our research.

B.2.6.3 Classification criteria

Three versions of classification criteria for individual headache attacks were designed in chronological order. The motivation for each new version follows from generating and analyzing the headache attack registrations and their classification results. This motivation will be further elaborated on in Section B.4. For the remainder of this appendix, note that the term "classification criteria" always refers to the criteria applied by the designed classification system, as opposed to the term "diagnostic criteria" which refers to the ICHD-3 criteria.

Version 1 of the classification criteria The first version of the classification criteria consists of exactly these ICHD-3 criteria that are targeted at individual attacks. To obtain them, all criteria that do not relate to an individual attack were put in a separate set (a). This set includes criteria targeted at the frequency, total number and/or time period of the attacks, as well as the caution criterion "Not better accounted for by another ICHD-3 diagnosis" which is a reminder to always consider other diagnoses that might better explain the headache. These criteria in set (a) can be ignored for the classification of an individual headache attack.

Version 2 of the classification criteria To properly assess the headache duration and characteristics, the ICHD-3 diagnostic criteria for *migraine without aura* and *CH* require the headache attack to be "untreated or unsuccessfully treated", and "untreated", respectively. As a consequence, (successfully) treated attacks are not taken into account during clinical diagnosis based on ICHD-3. Hence, they do not meet version 1 of the classification criteria. However, in practice, both migraine and CH patients treat their attacks with medication, and often with success. Therefore, in version 2, all classification criteria of version 1 are again applied strictly, except for the criteria related to medication treatment. Concretely, this means that for *migraine without aura*, it is no longer required that the attack is "untreated or unsuccessfully treated". Similarly, it is not required that an attack is "untreated" to be classified as a *CH* attack.

Version 3 of the classification criteria In close collaboration with headache experts, the following decisions were made in a 3rd version of the classification criteria:

- The ICHD-3 criteria for *migraine without aura* and *CH* state a required duration, which is conditioned on the attack being "untreated or unsuccessfully treated",

and "untreated", respectively. In version 2, the treatment conditions for both disorders are no longer required. However, (successfull) treatment may have an influence on the perceived duration of the attack and symptomatology. In fact, ICHD-3 does not define the required duration of a (successfully) treated attack. Hence, for version 3 of the criteria, the condition on the specified duration of an attack does also not need to be fulfilled.

- For *CH*, the ICHD-3 criteria state that during less than 50% of the active time course of a cluster period with attacks, attacks may be less severe and/or of shorter or longer duration, as compared to the diagnostic criteria. This is another reason to ignore the duration criterion in version 3 of the classification criteria. Moreover, specifically for the classification of *CH* attacks, it also leads to the decision to not require the fulfillment of the severity criterion.

Overview of the different versions of the classification criteria Table B.3 shows an overview of how the ICHD-3 diagnostic criteria are mapped to the classification criteria of *migraine without aura*, *CH* and *episodic TTH*. For each disorder type, the diagnostic criteria are divided in three different disjunctive sets:

(a) the set of criteria that are not targeted at characteristics of individual attacks – they are ignored by all versions of the classification criteria for individual headache attacks;

(b) the set of criteria targeted at characteristics of individual attacks, that *need* to be fulfilled *by version 3 of the classification criteria*;

(c) and the remaining set of criteria targeted at characteristics of individual attacks, that do *not* need to be fulfilled *by version 3 of the classification criteria*.

In the remainder of this appendix, parts of the criteria in set (c) are often referred to as the *treatment criterion*, *duration criterion* and *severity criterion*. Table B.4 makes explicit which parts of the criteria are exactly meant by those terms.

In summary, Table B.5 shows the concrete criteria that are required for the classification of individual headache attacks as the different types, in the different versions of the classification criteria. In addition, it also highlights for each version of the criteria, which individual criteria are additionally evaluated for each headache attack that is classified as that type. This will be further explained in the next section. To make the overview in Table B.5 clear, the table uses the names of the sets mentioned in Table B.3, and the criterion names made explicit in Table B.4.

B.2.6.4 Methodology of the classification system

This section presents the methodology of the semantic classification system for individual headache attacks.

Table B.3: Constructed sets of classification criteria based on ICHD-3 for the headache classification system. The table gives an overview of how the ICHD-3 criteria of migraine without aura (criteria 1.1), cluster headache (criteria 3.1) and episodic tension-type headache (criteria 2.1 for infrequent episodic tension-type headache), are split up in three sets for version 3 of the classification criteria for an individual headache attack: set (a) with criteria not targeted at individual attacks; set (b) with criteria targeted at individual attacks that are required to classify an attack as this type; and set (c) with criteria targeted at individual attacks that are not required to classify an attack as this type. The mentioned letters refer to the criteria letters as how they are presented in ICHD-3. Note that for version 1 of the classification criteria, both set (b) and set (c) are required for the classification of an individual attack as that type. For version 2, all criteria in set (b) and set (c) are required for classification, except for the treatment criteria (see Table B.4 and Table B.5).

Disorder type	Set	Criteria
Migraine without aura	Set (a)	A: At least five attacks 1 fulfilling criteria B–D E: Not better accounted for by another ICHD-3 diagnosis
	Set (b)	C: Headache has at least two of the following four characteristics: 1. unilateral location 2. pulsating quality 3. moderate or severe pain intensity 4. aggravation by or causing avoidance of routine physical activity D: During headache at least one of the following: 1. nausea and/or vomiting 2. photophobia and phonophobia
	Set (c)	B: Headache attacks lasting 4–72 hours (when untreated or unsuccessfully treated)
Cluster headache	Set (a)	A: At least five attacks fulfilling criteria B–D D: Occurring with a frequency between one every other day and eight per day E: Not better accounted for by another ICHD-3 diagnosis
	Set (b)	B (part): Unilateral orbital, supra-orbital and/or temporal pain C: Either or both of the following: 1. at least one of the following symptoms or signs, ipsilateral to the headache: a) conjunctival injection and/or lacrimation b) nasal congestion and/or rhinorrhoea c) eyelid oedema d) forehead and facial sweating e) miosis and/or ptosis 2. a sense of restlessness or agitation
	Set (c)	B (part): Severe or very severe pain lasting 15–180 minutes (when untreated)
Episodic tension-type headache	Set (a)	A: At least 10 episodes of headache occurring on <1 day/month on average (<12 days/year) and fulfilling criteria B–D E: Not better accounted for by another ICHD-3 diagnosis
	Set (b)	C: At least two of the following four characteristics: 1. bilateral location 2. pressing or tightening (non-pulsating) quality 3. mild or moderate intensity 4. not aggravated by routine physical activity D: Both of the following: 1. no nausea or vomiting 2. no more than one of photophobia or phonophobia
	Set (c)	B: Lasting from 30 minutes to 7 days

Table B.4: Mapping of criterion types to ICHD-3 criteria in set (c) of Table B.3. The table gives an overview of how the ICHD-3 criteria in set (c) (i.e., the criteria targeted at individual attacks that are not required to classify an attack as the corresponding type in version 3 of the classification criteria, see Table B.3) of the different disorder types are mapped onto the different criterion types: duration, treatment and severity. If set (c) does not contain the criterion type, the table cell is empty ('/').

Disorder type	Duration criterion	Treatment criterion	Severity criterion
Migraine without aura	Attack lasting 4–72 hours	Attack untreated or unsuccessfully treated	/
Cluster headache	Pain lasting 15–180 minutes	Attack untreated	Severe or very severe pain
Episodic tension-type headache	Pain lasting 30 minutes to 7 days	/	/

Table B.5: Required and additionally evaluated criteria for all versions of classification criteria for individual headache attacks. The table presents which ICHD-3 diagnostic criteria are used as actual required classification criteria in the different versions of the classification criteria for individual headache attacks of the different disorder types, and which criteria are additionally evaluated to enrich the classification output. To make this overview compact and clear, the names of the sets from Table B.3 and criterion names from Table B.4 are used.

	Required criteria	Additionally evaluated criteria
Version 1	set (b) & set (c)	/
Version 2	set (b) & set (c), excluding treatment criterion	treatment criterion
Version 3	set (b)	duration, treatment & severity criterion

Classification input The input of the classification system consists of the semantic description of a headache attack registered via the mBrain app with the mBrain ontology, of which an example is presented in Listing B.1.

Classification output The classification criteria are validated for each individual headache attack, independently from other events and independently per type. Hence, it is checked separately for each of the three disorder types whether the attack can be classified as that type. This means that the output of classifying an individual headache attack is a set of zero to three individual classifications.

One classification contains two things: (i) the type of disorder the attack is classified as, and (ii) a binary indication of fulfillment for each criterion that is targeted at an individual headache attack but not required for the classification, according to the considered version of the classification criteria. The criteria in this set are specified as "additionally evaluated criteria" in Table B.5. The rationale behind this is that it is still interesting for a person (e.g., a physician) to know whether the unrequired criteria are actually fulfilled or not; this gives more information than a classified type only, and can be important for treatment.

Classification process The classification criteria of each disorder type have been translated into a semantic query that is executed on a data store containing the mBrain ontology data and the semantic representation of the headache attack that needs to be classified. Moreover, the additionally evaluated criteria have also been translated into a set of additional simple queries, which need to be executed after the main classification query. Altogether, this results in one set of queries per considered headache disorder phenotype, which need to be executed in order.

In terms of technologies, all semantic data is represented in Resource Description Framework (RDF) format [57]. The SPARQL Protocol and RDF Query Language (SPARQL) is used to write and evaluate the queries [58]. To manage the query execution process, including the addition and deletion of events to the data store and the semantic reasoning, any semantic reasoning engine such as Apache Jena or RDFox [59] can be used.

In Listing B.2, an example of how the version 3 classification criteria are translated into the main SPARQL classification query is given for *migraine without aura*.

B.2.6.5 Knowledge-based diagnosis of headache disorders

Based on the classification of individual headache events experienced by a patient, a second step can be to also classify them as a specific disorder, as this is semantically equivalent to diagnosing that patient with a specific disorder. To do so, the same semantic classification system can be applied, with a set of disorder classification queries. The criteria that should be used for this in a first version, are those that are not targeted at individual headache attacks but specify the frequency and time period of the

Listing B.2: SPARQL query corresponding to version 3 of the classification criteria, that is used by the semantic headache classification system to filter headache attacks of type migraine without aura. Similar queries exist to classify CH attacks and episodic TTH episodes, and for other versions of the classification criteria.

```
PREFIX m: <http://contextaware.ilabt.imec.be/ontology/medical.owl#>
PREFIX h: <http://contextaware.ilabt.imec.be/ontology/headache.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
    ?a rdf:type h:MigraineWithoutAuraAttack ;
       h:hasMigraineWithoutAuraCharacteristics [ ] .
}
WHERE {
    ?a rdf:type h:HeadacheAttack .

    # headache has at least two of the following four characteristics:
    # 1. unilateral location
    # 2. pulsating quality
    # 3. moderate or severe pain intensity
    # 4. aggravation by or causing avoidance of routine physical
    #    activity (e.g. walking or climbing stairs)
    {
        SELECT ?a (COUNT(?b) AS ?numberOfCriteria)
        WHERE  {
            {
                ?a h:isUnilateral ?b .
                FILTER(xsd:boolean(?b) = xsd:boolean("true"))
            } UNION {
                ?a m:hasAssociatedMedicalSymptom ?b .
                ?b rdf:type h:PulsatingPain .
            } UNION {
                ?a m:hasPainIntensity ?b .
                { ?b rdf:type h:Moderate }
                UNION
                { ?b rdf:type h:Severe }
            } UNION {
                SELECT DISTINCT ?a ?b
                WHERE {
                    ?a m:hasID ?b .
                    ?a m:hasAssociatedMedicalSymptom ?c .
                    { ?c rdf:type m:MovementSensitivity }
                    UNION
                    { ?c rdf:type h:PainAggravationByRoutinePhysicalActivity }
                }
            }
        }
        GROUP BY ?a HAVING ( ?numberOfCriteria >= 2 )
    }

    # during headache at least one of the following:
    # 1. nausea and/or vomiting
    # 2. photophobia and phonophobia
    FILTER EXISTS {
        {
            { ?a m:hasAssociatedMedicalSymptom [ rdf:type m:Vomitus ] }
            UNION
            { ?a m:hasAssociatedMedicalSymptom [ rdf:type m:Nausea ] }
        } UNION {
            ?a m:hasAssociatedMedicalSymptom
                [ rdf:type m:Photophobia ], [ rdf:type m:Phonophobia ]
        }
    }
}
```

attacks that fulfill the other ICHD-3 criteria. This corresponds to the set of criteria that were omitted for the classification of individual headache attacks, i.e., the criteria in set (a) of each disorder type in Table B.3.

B.2.7 Knowledge-based detection of headache triggers

This section discusses the design of a preliminary knowledge-based trigger detection system evaluating triggers for headache at a personal level. In headache medicine, certain triggers are well-known (e.g., menstrual cycle in women with migraine, alcohol in CH patients), but others are debated. The question often remains whether certain events, behaviors or external factors can truly be classified as triggering the attack, or as premonitory (prodromal) phenomena of the attack. In clinical practice, physicians and even patients find it hard to disentangle this question and leave the suggestion that certain symptoms may be misattributed by patients as triggers, but basically are headache associated symptoms at the start of the attack [60, 61].

A preliminary knowledge-based trigger detection system could – given the correct knowledge about triggers and an accurate detection of them using the collected data – potentially be a valuable tool for patients with regular headache attacks. Such a system takes advantage of the wide range of contextual data collected in the mBrain study. In the mBrain app, upon the registration of a headache attack, patients can select any possible triggers for that attack out of a list. This list, which is specified in Table B.2, originates from medical expert knowledge on headache attacks, and especially migraine and CH [62, 63]. For each trigger in this list, the question is whether the occurrence of this trigger can be detected based on the data collected in the mBrain study. Based on the automatically generated activity, stress and sleep events and the collected contextual data, the occurrence of five headache triggers out of the provided trigger list could potentially be detected: physical exercise, sleep deprivation, stress, relieve from stress, and skipping of meals.

For a semantic trigger detection system to actually detect the occurrence of any of these events/situations, queries should be written that can be automatically executed on a data window of specified duration. This window should contain all physiological and contextual data and events of interest that are collected and generated within its boundaries. This is where the mBrain ontology again has an important role: it provides a means to semantically describe and link all this data in a common, machine-interpretable format. The size of the data window also needs to be defined dynamically based on medical expert knowledge, taking into account information about patient and context, as this makes the assumption that the trigger lies within this time range. If a query detects that a known trigger for a patient has occurred, it could for example generate an alarm that a headache attack might follow for that patient, which could be translated into a mobile notification of the mBrain app.

Listing B.3: Example query that can be used in a knowledge-based trigger detection system to detect the occurrence of a physical exercise trigger, and generate an alarm of a potential upcoming headache attack if this trigger is known and detected. The query is executed periodically on a sliding data window of which the size (time duration) can be defined dynamically based on patient and context.

```
PREFIX m: <http://contextaware.ilabt.imec.be/ontology/medical.owl#>
PREFIX h: <http://contextaware.ilabt.imec.be/ontology/headache.owl#>
PREFIX d: <http://IBCNServices.github.io/DAHCC/>
PREFIX s: <https://saref.etsi.org/saref4ehaw/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX time : <http://www.w3.org/2006/time#>

CONSTRUCT {
    _:a rdf:type h:HeadacheAlarm ; h:relatedToTrigger ?t ; m:targetedAt ?p .
}
FROM STREAM <http://contextaware.ilabt.imec.be/stream> [RANGE 3 HOUR STEP 1 HOUR]
FROM <http://contextaware.ilabt.imec.be/trigger-knowledge.rdf>
WHERE {
    # consider a given patient
    ?p rdf:type s:Patient .

    # physical exercise is a known headache trigger for that patient
    ?p h:hasHeadacheTrigger ?t . ?t rdf:type m:PhysicalExercise .

    # patient has performed a physical exercise activity in the given time window
    # for at least 5 minutes (= 300 seconds) (physical exercise = cycling or running)
    ?p s:hasActivity [ rdf:type d:PhysicalExercise ] ; s:activityDuration ?d .
    FILTER (xsd:float(?d) >= xsd:float(300))
}
LIMIT 1
```

In Listing B.3, a very simple illustrative example is given of a stream processing query that could detect the physical exercise trigger. It checks whether physical exercise is a known trigger for any existing patient in the data store, and if so, whether any activity representing physical exercise has been detected for this patient with a duration of at least 5 minutes in the considered time window. If this is the case, a headache alarm is generated.

B.3 Results

During the period from July 2020 until October 2020, 7 patients participated in the first data collection wave of the mBrain study: 5 migraine patients and 2 CH patients. In the second data collection wave, which took place from October 2020 until January 2021, 11 more patients participated: 9 migraine patients and 2 CH patients. First wave participants used the mBrain v1 application, while second wave participants used mBrain v2. All CH patients were chronic CH patients (ICHD-3 3.1.2). This section presents the first results of the mBrain study.

Table B.6: Demographics and baseline characteristics of the group of mBrain participants. Numbers are presented for the total group, as well as separately for the migraine and cluster headache patients in the participant group. The results are provided for wave 1 and wave 2 combined. Abbreviations used in this table: pct. is percentage, SD is standard deviation, IQR is interquartile range, '#' represents 'number of'.

	Total group (n=18)	**Migraine (n=14)**	**Cluster headache (n=4)**
Age (years), mean (SD)	39.1 (11.9)	37.1 (10.6)	46 (15.4)
Sex (female), pct. (ratio)	66.7% (12/18)	85.7% (12/14)	0% (0/4)
Migraine days per month, mean (SD)	/	5.2 (2.2)	/
Days with headache attacks per month, mean (SD)	/	/	23.8 (7.5)
Weight (kg), mean (SD)	74.3 (13.5)	70.0 (11.1)	89.2 (10.4)
Height (cm), mean (SD)	173 (8.5)	170 (6.9)	182 (5.6)
# children, median (IQR)	1 (0–2)	1 (0–2)	1 (0–1)
Alcoholic beverages per week (units), mean (SD)	2.3 (2.9)	1.7 (1.9)	4.3 (5.1)
Cigarettes per day (units), median (IQR)	0 (0–2)	0 (0–0)	9 (6–11)

B.3.1 Data collection results

Table B.6 describes the general demographic characteristics of the study subjects. Table B.7 presents their current and previous acute and preventive medication use.

Table B.8 shows the general statistics of the first and second data collection waves. It zooms in on the data collection, timeline activity & interaction, and daily records.

In Table B.9, statistics are shown about the headache attacks registered during the two data collection waves, split up based on the diagnosis of the participants. They compare the intensity, duration, location, symptoms and triggers of the attacks.

B.3.2 Knowledge-based headache classification results

The proposed knowledge-based classification system for headache attacks, presented in Section B.2.6, has been applied on the headache data collected during the two data collection waves.

Table B.10 focuses on the migraine patients and shows the number of classifications of the attacks experienced by the migraine patients as *migraine without aura*, *CH* and *episodic TTH*, using the three versions of the classification criteria. It shows the results for both the first data collection wave (n = 20 attacks) and the second wave (n = 78 attacks). A similar overview is shown in Table B.11 for the CH patients in the first wave (n = 20 attacks) and second wave (n = 15 attacks).

Table B.12 further details the results of applying version 3 of the classification criteria on the registered headache attacks. It also shows how often the additionally evaluated criteria are fulfilled for the classifications of each type. Moreover, it

Table B.7: Current and previous use of medications for headache disorder in the group of mBrain participants. Numbers are presented for the total group, as well as separately for the migraine and cluster headache patients in the participant group. The results are provided for wave 1 and wave 2 combined. Abbreviations used in this table: pct. is percentage, IQR is interquartile range, '#' represents 'number of'.

	Total group (n=18)	**Migraine (n=14)**	**Cluster headache (n=4)**
Current use of acute medication, pct.	100%	100%	100%
Current # acute medications in use, median (IQR)	3 (2–3)	3 (2–3)	3 (2–4)
Current use of preventive medication, pct.	88.9%	85.7%	100%
Current # preventive medications in use, median (IQR)	1 (1–2)	1 (1–1)	1 (1–1)
# previous acute medications used, median (IQR)	2 (1–3)	2 (1–4)	0 (0–1)
# previous preventive medications used, median (IQR)	2 (1–4)	1 (1–4)	1 (1–3)

makes an analysis of whether the diagnosis of the migraine and CH patients corresponds to the disorder type for which the highest number of headache attacks is classified as that type, as compared to the other types. We refer to this as patients with "mostly diagnosis classifications".

B.3.3 Knowledge-based trigger detection results

A knowledge-based headache trigger detection system, following the methodology presented in Section B.2.7, offers the tools to detect the occurrence of some known triggers based on the data collected in the mBrain study. Currently, the only source of knowledge about headache attack triggers is the set of triggers indicated by the patient himself. As a first step to investigate the potential of a knowledge-based trigger detection system, this section checks with two example cases whether indicated, detectable triggers can actually be backed up with the collected physiological data, i.e., whether one can find proof in the data that the trigger actually occurred in the period preceding the headache attack. In other words, this is equivalent to investigating whether a query-based system would have been able to detect the trigger for the attack.

The first example case involves the "physical exercise" trigger. In a simple set-up (see Listing B.3), this trigger could be detected by checking if the patient has performed any activity representing physical exercise in the considered time window. Given the activity types detectable by the ML algorithms, this includes any activity of type running or cycling (excluding commuting, since our definition of commuting requires no real physical exercise). Out of all mBrain patients, 2 migraine patients have indicated

Table B.8: General statistics of data collection during the first and second mBrain data collection wave. Statistics mentioning "(pp)" are "per patient" statistics: they are first calculated per patient, and then aggregated over all patients. Other abbreviations used in this table: aut. is automatically added, man. is manually added, pct. is percentage, pds. is periods, w/ is with, w/o is without, '#' represents 'number of'.

	Wave 1	**Wave 2**
# patients	7	11
Duration of trial (days) (pp), mean (SD)	22.57 (1.62)	22.09 (0.30)
Data collection statistics		
Connected Empatica time per trial day (HH:mm) (pp), mean (SD)	09:06 (03:54)	12:49 (04:49)
# location points per trial day (pp), mean (SD)	91.49 (151.62)	109.78 (83.84)
# tags per trial day (pp), mean (SD)	0.41 (0.56)	0.89 (1.07)
Timeline activity & interaction statistics		
Headache attacks		
# headache attacks (pp), mean (SD)	5.71 (4.68)	8.45 (3.14)
Pct. of patients w/o any headache attack	14.29%	0.00%
Medicine intakes		
# medicine intakes (pp), mean (SD)	3.86 (5.67)	5.64 (3.38)
Pct. of patients w/o any medicine intake	57.14%	9.09%
Activities		
# man. activities (pp), mean (SD)	1.07 (0.46)	2.60 (1.77)
# aut. activities per trial day (pp), mean (SD)	35.81 (12.71)	46.24 (16.16)
Pct. of aut. activities of sedentary type (pp), mean (SD)	90.42% (3.47%)	88.29% (6.22%)
Pct. of aut. activities *fully* confirmed (pp), mean (SD)	19.19% (26.72%)	45.90% (30.46%)
Pct. of aut. activities *only* confirmed as sedentary, but w/o explicit confirmation of predicted type (pp), mean (SD)	0.00% (0.00%)	12.01% (14.83%)
Pct. of aut. activities w/ *only* corrected time (pp), mean (SD)	0.04% (0.08%)	0.56% (0.64%)
Pct. of aut. activities w/ corrected type (pp), mean (SD)	6.34% (6.83%)	5.31% (6.03%)
Pct. of aut. activities removed (pp), mean (SD)	3.22% (7.42%)	10.27% (8.31%)
Pct. of aut. activities ignored (pp), mean (SD)	71.20% (37.99%)	25.94% (33.73%)
Sleep periods		
# man. sleep pds. (pp), mean (SD)	13.86 (5.70)	19.45 (6.74)
# aut. sleep pds. (pp), mean (SD)	31.57 (9.91)	37.36 (13.17)
Pct. of aut. sleep pds. confirmed (pp), mean (SD)	6.11% (9.98%)	13.46% (13.16%)
Pct. of aut. sleep pds. w/ corrected time (pp), mean (SD)	0.48% (1.26%)	2.62% (3.76%)
Pct. of aut. sleep pds. corrected to activity (pp), mean (SD)	0.00% (0.00%)	0.70% (2.32%)
Pct. of aut. sleep pds. removed (pp), mean (SD)	5.46% (8.33%)	23.43% (28.16%)
Pct. of aut. sleep pds. ignored (pp), mean (SD)	87.96% (16.08%)	59.79% (30.52%)
Stress periods		
# man. stress pds. per trial day (pp), mean (SD)	0.04 (0.08)	0.05 (0.11)
# aut. stress pds. per trial day (pp), mean (SD)	15.86 (11.30)	6.66 (2.84)
Pct. of aut. stress pds. confirmed w/o level (pp), mean (SD)	2.63% (5.92%)	0.00% (0.00%)
Pct. of aut. stress pds. confirmed w/ level 1/2 (pp), mean (SD)	1.48% (3.70%)	44.89% (32.41%)
Pct. of aut. stress pds. corrected w/ level 0 (pp), mean (SD)	17.49% (29.45%)	28.62% (28.03%)
Pct. of aut. stress pds. removed (pp), mean (SD)	5.00% (6.17%)	2.66% (2.58%)
Pct. of aut. stress pds. w/ corrected time (pp), mean (SD)	0.03% (0.07%)	0.04% (0.14%)
Pct. of aut. stress pds. ignored (pp), mean (SD)	73.37% (36.33%)	23.79% (34.77%)
Daily record (DR) statistics		
Pct. of trial days w/ part of DR provided (pp), mean (SD)	78.89% (19.00%)	92.99% (6.55%)
Pct. of trial days w/ daily stress level provided (pp), mean (SD)	73.07% (25.05%)	91.34% (6.59%)
Pct. of trial days w/ daily mood provided (pp), mean (SD)	73.66% (23.75%)	91.34% (6.59%)
Pct. of trial days w/ daily food intake provided (pp), mean (SD)	60.17% (28.33%)	87.62% (6.15%)
Pct. of trial days w/ DR fully completed (pp), mean (SD)	58.87% (30.68%)	87.62% (6.15%)

Table B.9: Statistics of registered headache attacks during the first and second mBrain data collection wave. The statistics are split up based on the diagnosis of the participating patients. Columns entitled W1 and W2 present the results for wave 1 and wave 2, respectively. Abbreviations used in this table: pct. is percentage, SD is standard deviation, w/ is with, w/o is without, '#' represents 'number of'.

	Migraine		**Cluster headache**	
	W1	**W2**	**W1**	**W2**
# patients	5	9	2	2
Duration of trial (days), mean (SD)	22.60	22.11	22.50	22.00
	(1.95)	(0.33)	(0.71)	(0.00)
Total # headache attacks	20	78	20	15
# attacks per patient, mean (SD)	4.00	8.67	10.00	7.50
	(2.74)	(3.46)	(7.07)	(0.71)
Pain intensity of attacks, mean (SD)	2.00	1.76	1.50	2.27
	(0.92)	(0.81)	(0.61)	(0.70)
Pct. of attacks w/ pain intensity:				
0 (no pain)	0.00%	0.00%	0.00%	0.00%
1 (mild pain)	30.00%	46.15%	55.00%	13.33%
2 (moderate pain)	50.00%	33.33%	40.00%	46.67%
3 (severe pain)	10.00%	19.23%	5.00%	40.00%
4 (very severe pain)	10.00%	1.28%	0.00%	0.00%
Duration (HH:mm), mean (SD)	08:27	05:59	00:22	00:50
	(09:39)	(05:42)	(00:15)	(00:20)
Pct. of unilateral attacks	45.00%	73.08%	100.00%	100.00%
# symptoms per attack, mean (SD)	4.40	2.29	0.65	2.73
	(2.41)	(2.10)	(0.67)	(1.03)
# triggers per attack, mean (SD)	0.75	0.83	0.35	0.33
	(0.79)	(0.90)	(0.49)	(0.49)
Pct. of attacks w/o symptom	0.00%	19.23%	45.00%	0.00%
Pct. of attacks w/o trigger	45.00%	43.59%	65.00%	66.67%
Pct. of attacks treated	95.00%	51.28%	70.00%	100.00%
Pct. of attacks treated successfully	75.00%	37.18%	70.00%	93.33%
Pct. of attacks w/ selected symptom:				
conjunctival injection	10.00%	0.00%	5.00%	26.67%
lacrimation	15.00%	5.13%	0.00%	26.67%
ptosis	0.00%	8.97%	0.00%	0.00%
miosis	0.00%	6.41%	0.00%	0.00%
eyelid oedema	5.00%	6.41%	10.00%	0.00%
nasal congestion	20.00%	7.69%	0.00%	80.00%
rhinorrhoea	20.00%	3.85%	0.00%	6.67%
sweaty forehead and face	25.00%	1.28%	0.00%	0.00%
pulsating pain	75.00%	34.62%	45.00%	53.33%
movement sensitivity	40.00%	28.21%	0.00%	6.67%
pain increment during routine physical activity	40.00%	25.64%	0.00%	6.67%
restlessness or agitation	30.00%	21.79%	5.00%	60.00%
photophobia	95.00%	24.36%	0.00%	6.67%
phonophobia	35.00%	29.49%	0.00%	0.00%
osmophobia	20.00%	2.56%	0.00%	0.00%
nausea	10.00%	23.08%	0.00%	0.00%
vomiting	0.00%	0.00%	0.00%	0.00%

Table B.10: Results of applying the classification criteria on registered headache attacks of the migraine participants. These results are the output of applying the different versions of the classification criteria for individual headache attacks on the headache attacks experienced by the 14 migraine patients that have successfully participated in the first and second wave of the mBrain study (5 in wave 1, 9 in wave 2). Columns entitled W1 and W2 present the results for wave 1 and wave 2, respectively. The numbers in the cells represent the number of attacks that are classified as the type specified in the row header, out of the total number of attacks registered by migraine patients during that wave (i.e., 20 attacks for wave 1, 78 attacks for wave 2); the percentages of these ratios are given between brackets. Abbreviations used in this table: CH is cluster headache, TTH is tension-type headache, w/o is without.

	Criteria v1		Criteria v2		Criteria v3	
Classification	**W1**	**W2**	**W1**	**W2**	**W1**	**W2**
Migraine w/o aura	1	7	5	13	7	21
	(5.00%)	(8.97%)	(25.00%)	(16.67%)	(35.00%)	(26.92%)
CH	0	0	0	0	4	15
	(0.00%)	(0.00%)	(0.00%)	(0.00%)	(20.00%)	(19.23%)
Episodic TTH	10	45	10	45	11	46
	(50.00%)	(57.69%)	(50.00%)	(57.69%)	(55.00%)	(58.97%)

Table B.11: Results of applying the classification criteria on registered headache attacks of the cluster headache participants. These results are the output of applying the different versions of the classification criteria for individual headache attacks on the headache attacks experienced by the 4 CH patients that have successfully participated in the first and second wave of the mBrain study (2 in wave 1, 2 in wave 2). Columns entitled W1 and W2 present the results for wave 1 and wave 2, respectively. The numbers in the cells represent the number of attacks that are classified as the type specified in the row header, out of the total number of attacks registered by CH patients during that wave (i.e., 20 attacks for wave 1, 15 attacks for wave 2); the percentages of these ratios are given between brackets. Abbreviations used in this table: CH is cluster headache, TTH is tension-type headache, w/o is without.

	Criteria v1		Criteria v2		Criteria v3	
Classification	**W1**	**W2**	**W1**	**W2**	**W1**	**W2**
Migraine w/o aura	0	0	0	0	0	0
	(0.00%)	(0.00%)	(0.00%)	(0.00%)	(0.00%)	(0.00%)
CH	0	0	0	6	3	14
	(0.00%)	(0.00%)	(0.00%)	(40.00%)	(15.00%)	(93.33%)
Episodic TTH	2	12	2	12	19	13
	(10.00%)	(80.00%)	(10.00%)	(80.00%)	(95.00%)	(86.67%)

Table B.12: Results of applying version 3 of the classification criteria on all registered headache attacks. The table presents the results of applying version 3 of the classification criteria for individual headache attacks on the 133 headache attacks experienced by the 18 patients that have successfully participated in the first and second data collection wave of the mBrain study, split up based on the diagnosis of the participating patients. Columns entitled W1 and W2 present the results for wave 1 and wave 2, respectively. Patients with mostly diagnosis classifications are patients for who the patient's diagnosis matches the disorder type for which there are the most classifications out of the patient's headache attacks. Abbreviations used in this table: classif. is classifications, CH is cluster headache, TTH is tension-type headache, pct. is percentage, SD is standard deviation, '#' represents 'number of'.

	Migraine		**Cluster headache**	
	W1	**W2**	**W1**	**W2**
# patients	5	9	2	2
Total # headache attacks	20	78	20	15
# patients with mostly diagnosis classif.	2	2	0	2
(pct. of # patients)	(40.00%)	(22.22%)	(0.00%)	(100.00%)
# patients with mostly diagnosis classif., not considering *episodic TTH* classif.	3	6	1	2
(pct. of # patients)	(60.00%)	(66.67%)	(50.00%)	(100.00%)
Classif. of attacks as *migraine without aura*				
# classif. (pct. of # headache attacks)	7 (35.00%)	21 (26.92%)	0 (0.00%)	0 (0.00%)
# classif. with fulfilled duration criterion	5	13	0	0
# classif. with fulfilled treatment criterion	1	9	0	0
# classif. with all criteria fulfilled	1	7	0	0
# registered symptoms per classification, mean (SD)	6.86 (1.77)	4.57 (2.40)	/	/
Classif. of attacks as *CH*				
# classif. (pct. of # headache attacks)	4 (20.00%)	15 (19.23%)	3 (15.00%)	14 (93.33%)
# classif. with fulfilled duration criterion	2	3	3	14
# classif. with fulfilled treatment criterion	0	5	0	0
# classif. with fulfilled severity criterion	0	5	0	6
# classif. with all criteria fulfilled	0	0	0	0
# registered symptoms per classification, mean (SD)	7.50 (1.29)	4.07 (2.81)	1.67 (0.58)	2.86 (0.95)
Classif. of attacks as *episodic TTH* episodes				
# classif. (pct. of # headache attacks)	11 (55.00%)	46 (58.97%)	19 (95.00%)	13 (86.67%)
# classif. with fulfilled duration criterion	10	45	2	12
# registered symptoms per classification, mean (SD)	2.73 (1.27)	1.57 (0.81)	0.63 (0.68)	2.54 (0.88)
# classif. per attack				
# attacks with 0 classif.	1	9	1	0
(pct. of # headache attacks)	(5.00%)	(11.54%)	(5.00%)	(0.00%)
# attacks with 1 classif.	16	56	16	3
(pct. of # headache attacks)	(80.00%)	(71.79%)	(80.00%)	(20.00%)
# attacks with 2 classif.	3	13	3	12
(pct. of # headache attacks)	(15.00%)	(16.67%)	(15.00%)	(80.00%)
# attacks with 3 classif.	0	0	0	0
(pct. of # headache attacks)	(0.00%)	(0.00%)	(0.00%)	(0.00%)

"physical exercise" as a trigger for a total of 4 headache attacks. For 2 headache attacks, a running event has actually taken place in the period of 3 hours before the attack. The details of one example are given below:

```
5:40 PM - 6:42 PM: Running activity
7:15 PM - 10:10 PM: Headache attack
```

A second example case investigates the "stress" trigger. Stress as a trigger requires more knowledge about the type and period of stress, to write an accurate query to detect it. Nevertheless, the number of stress events is evaluated for the 11 headache attacks with stress as indicated trigger, experienced by 3 migraine patients. For 6 of the 11 attacks, at least one confirmed stress event was observed in the period of 5 hours before the attack. For 4 of those, this was also the case in the hour preceding the attack. When combining multiple stress events and observing their total duration in a certain period, longer periods of stress become visible preceding some attacks for one specific patient. A good example to illustrate this is the following:

```
8:46 AM - 8:47 AM: Stress period
8:52 AM - 8:55 AM: Stress period
9:20 AM - 9:33 AM: Stress period
9:47 AM - 9:54 AM: Stress period
10:03 AM - 10:11 AM: Stress period
10:56 AM - 8:03 PM: Headache attack
```

B.4 Discussion

Based on the results of the first & second data collection wave of the mBrain study presented in Section B.3, several aspects can be discussed with respect to the objectives of this appendix outlined in Section B.1.4.

B.4.1 Knowledge-based classification of individual headache attacks

Headache registrations As can be observed in Table B.9, 98 headache attacks have been registered by the migraine patients that participated in the mBrain study, and 35 attacks by the CH patients. This leads to a total of 133 registered attacks.

Looking at the attacks of the first data collection wave, a first observation was the fact that sometimes, no symptoms and/or triggers were selected by the patient. This information was not required in mBrain v1. In separate headache events, it is possible that a patient does not experience any of the symptoms in the list, or believes no item in the trigger list was a probable trigger for the attack. Patients with migraine or CH may also experience episodes of TTH, which is characterized by the absence of certain migraine or CH specific symptoms (see the ICHD-3 criteria for TTH, Table B.3). In

the analysis of the headache attacks, all unselected information is implicitly assumed to be not applicable. In mBrain wave 1, this assumes that patients process all available symptoms and triggers, and *only* select none, *if* none are applicable. It is important that this assumption is true, especially related to the selection of the symptoms, since those variables are used as input for the headache classification. However, from the collected participant feedback, it appeared that some patients simply did not always check (part of) the list of symptoms and triggers, because of lack of interest or time. Hence, in mBrain v2, the requirement was added that a patient needs to select at least one symptom and trigger when registering a headache attack, with the option at the bottom to select "none of those", which then functions as the new ground truth entry for absence of headache-associated symptomatology or trigger for the particular attack.

Surprisingly, the results on the headache attacks registered during the second wave, do not show a direct impact of these adaptations for migraine patients: the percentage of attacks without a trigger remained constant (45% in wave 1 vs. 43.59% in wave 2), while the percentage of attacks without a symptom increased with almost 20% (0% in wave 1 vs. 19.23% in wave 2). On the contrary, in patients with CH, the adaptations of mBrain v2 resulted in no attacks without symptoms versus 45% of attacks in wave 1 patients (mBrain v1). It is important to readdress here that the lack of symptoms or triggers in wave 2 is explicit: for this the participants had to explicitly select the option "none of those" at the bottom of the list with available options. Moreover, while the average number of triggers per attack slightly increased from 0.75 to 0.83 for migraine patients, the average number of symptoms per attack even decreased from 4.40 to 2.29. This does not infer any concrete conclusion, since the selection of symptoms is still not explicit on an individual per-symptom basis. What can be learned from our experience is that explicit information about attack symptomatology or triggers is necessary in further app development.

The general statistics on headache attack registrations by migraine and CH patients seem to confirm existing knowledge about both disorder types. First, the number of attacks during a trial period of comparable length is higher for CH. Second, the attacks of CH patients are shorter. Third, the attack semeiology for both disorders is in concordance with medical literature (e.g., more restlessness and cranial autonomic symptoms (CAS) in patients with CH versus more hypersensitivity symptoms and nausea in patients with migraine). Fourth, if we calculate the total percentage of unilateral attacks, the attacks of CH patients are always unilateral, compared to only approximately 61% of the attacks of migraine patients.

An interesting observation, which does not correlate with the ICHD-3 criteria, is that the average intensity of attacks is not higher for CH patients compared to migraine patients, despite the fact that CH attacks are considered to be one of the most severe experiences of pain humans may have. A possible explanation was provided by a CH participant who stated that his or her assessment of the severity of attacks is subjective: due to desensitization, the patient assesses the severity lower compared to

the period recently after the attacks started and the diagnosis was made. The observation that some CH patients label the severity of certain attacks as mild or moderate has already been documented by other authors [64, 65]. In fact, some migraine patients with attacks of mild pain intensity also mentioned this subjective assessment during the outtake visit, although attacks in migraine patients may also represent close to the phenotypical form of TTH and the final moments of a migraine attack may resemble more characteristics fitting the TTH criteria [66]. Lastly, it can be observed that the number of registered symptoms is lower for CH patients compared to migraine patients. However, both CH patients in wave 1 confirmed they did not (always) check all symptoms in the list when registering a headache.

The inquiry of headache symptoms and triggers in a smartphone application system is an example of finding the balance between not having too big of an impact on the participant's daily life and routines, and making sure that the received information is as explicit as possible. The way that the information was requested in mBrain v1 was too focused on low intrusion, causing low information explicitness. Ideally, for every relevant symptom, the participant should indicate whether it is applicable or not with an explicit yes or no question. However, this would lean too much to the other side of the balance, potentially causing patients to not register any headache attacks as it becomes too time-consuming. Therefore, the changes in mBrain v2 try to find the right balance somewhere in the middle.

Classifications Closely analyzing the classifications of the headache attacks registered by the participants of both data collection waves, some general observations can be made, as well as noteworthy findings about the specific disorders.

A first and important observation in all versions of the classification criteria, is the high number of headache attacks that are classified as *episodic TTH*. From the results in Table B.12 we can calculate that with version 3 of the criteria, 89 out of 133 attacks (66.92%) receive this classification. 69 of those attacks (77.53%) also fulfill the required duration of 30 minutes to 7 days, while the other 20 attacks (22.47%) are all shorter than 30 minutes. However, a disclaimer should be made here. In version 3 of the classification criteria for *episodic TTH* in Table B.3 (set (b)), many criteria require the *absence* of a certain symptom: *no* pulsating pain, *not* aggravated by routine physical activity, *no* nausea, *no* vomiting, *not* both photophobia and phonophobia. These symptoms are all present in the list of selectable symptoms upon the registration of a headache attack, as indicated in Table B.2. As explained before, all unselected symptoms are implicitly considered to be non-applicable. Hence, a headache attack without any selected symptom will automatically fulfill the required classification criteria of *episodic TTH*. For version 1 and 2 of the criteria, this is true if the duration criterion is also fulfilled. In wave 1, selecting at least one symptom or selecting "none of those" was not yet required, and even in wave 2, no explicit "yes" or "no" answer is required for each individual symptom's presence. Hence,

incomplete registrations can lead to wrong classifications of *episodic TTH*. In contrast to *episodic TTH*, all symptom-related required classification criteria for *migraine without aura* and *CH* rely only on the *presence* of symptoms, which is always explicit. This is true for all versions of the classification criteria.

In general, observing the results of applying version 1 of the classification criteria to the registered headache attacks, the answer to the question whether the ICHD-3 diagnostic criteria can be strictly applied as classification criteria for individual attacks in a continuous follow-up setting, seems to be negative. In addition to the high number of attacks classified as *episodic TTH* episodes, the number of classifications as *migraine without aura* and *CH* is low: we can calculate from the results in Table B.10 that out of all 98 headache attacks experienced by migraine patients, only 8 are classified as *migraine without aura*, and no attack as *CH*. For the 35 attacks experienced by CH patients, the results are even worse: no attack receives any of both classifications. This leads to the conclusion that this version of the algorithm is neither sensitive nor specific for the discriminatory task between migraine without aura and CH attacks on the one hand and TTH episodes on the other hand.

An important reason for this observation is the treatment criterion, which is often not fulfilled in patients who have acute headache treatment in place. Concretely, we can calculate from the statistics in Table B.9 that approximately 66% of all attacks are treated and therefore could never be classified as *CH* if the treatment criterion would be taken into account. For *migraine without aura*, we can calculate that this number is 54%, since approximately 54% of all attacks are *successfully* treated. It is important to clarify that version 1 of the classification criteria was designed to strictly follow ICHD-3 definitions of headache attacks, even though ICHD-3 criteria for migraine and CH were designed to diagnose headache syndromes by analyzing multiple historic individual untreated attacks. There are currently no formal separate criteria in ICHD-3 for individual headache attacks only, in addition to the classification of disorders.

As such, the above discussed results from version 1 of the classification criteria confirm the rationale to test the exclusion of the treatment criterion in version 2 of the classification criteria. Observing the results of version 2 of the classification criteria, we can calculate that the number of *migraine without aura* classifications for attacks of migraine patients improve from 8 to 18. For CH patients, 6 out of the 35 attacks (17.14%) now receive the classification of *CH*. Because these numbers are still quite low, the criteria were further refined into version 3 of the criteria.

The main change to version 3 of the classification criteria was to also exclude the duration criterion, because of two reasons. First, ICHD-3 does not specify the required duration of a (successfully) treated attack. Second, the exact duration of an attack is often not *as* important as other location-related and symptom-related criteria. Observing the data, it is indeed true that there are attacks that fulfill those requirements, but not the duration requirement.

Taking a closer look at the classifications of the attacks of migraine patients with version 3 of the classification criteria, it can be calculated that only 28 of the 98 attacks (28.57%) are actually classified as *migraine without aura*. For this disorder, the required set of classification criteria in Table B.3 consists of two distinct criteria, annotated with letter C and D as in ICHD-3. While 59 out of 98 attacks (60.20%) fulfill criterion C, only 34 of the 98 attacks (34.69%) fulfill criterion D. Hence, criterion D is the main limiting factor to not have the remaining attacks be classified as *migraine without aura*. This criterion D requires associated nausea, vomiting, or the combination of photophobia and phonophobia. As can be calculated from the results in Table B.9, 38 out of 98 attacks (38.78%) have associated photophobia, 30 (30.61%) have phonophobia, 20 (20.41%) have nausea, and no attacks have associated vomiting. Hence, the low number of *migraine without aura* classifications is mainly caused by the lack of nausea or vomiting, and by the fact that often only one of photophobia or phonophobia occurs instead of both together. From the diagnostic criteria of ICHD-3, an attack fulfilling all but one criteria of *migraine without aura* only implies the diagnosis of *probable migraine without aura*, given no other ICHD-3 diagnosis is better accounted for. In general, this shows the difficulty that ICHD-3 has with capturing the intra-individual heterogeneity of migraine attacks into one set of criteria. This difficulty has an obvious impact on a system that assesses every attack individually, as compared to making a diagnosis based on a series of attacks. Therefore, further improving this classification system should consist of looking for techniques to incorporate these differences and trying to also integrate the *probable* disorder criteria in some way.

Interestingly, with version 3 of the classification criteria, we can calculate that 19 attacks of migraine patients are classified as *CH*. Observing the classification criteria of *CH*, two main components can be distinguished: (i) the pain should be unilateral around the orbit or temple, and (ii) the pain should have at least one associated symptom out of a given set. Because of the high intra-individual heterogeneity of migraine attacks, the location criteria (i) are sometimes fulfilled: 66 out of 98 migraine attacks (67.35%) were unilateral, and 56 of them (84.85%) were in the orbital, supra-orbital or frontal head region as well. The set of symptoms (ii) consists of restlessness/agitation, and the CAS. Previous research has shown that migraine and CH share common features in both the ICHD-3 criteria and semeiological descriptions [67–69], and that CAS regularly occur as symptoms of migraine attacks, even though they are not included in the ICHD-3 diagnostic criteria of migraine disorders [68]. Indeed, 34 of the 98 attacks experienced by the migraine participants had associated CAS. However, this previous research also shows that these CAS associated to migraine attacks are often bilateral, less severe, unrelated to the headache side, and less consistent with the headache attacks. Currently, the mBrain app allows no specification of such symptom characteristics. This would be especially relevant for the headache side, since the ICHD-3 criteria for *CH* require the CAS to be ipsilateral to the headache. Currently,

it is implicitly considered that this is the case in the classification process. Future improvement of the headache registration process should therefore include the explicit request of this information. This will be especially useful for migraine patients who often experience unilateral headache attacks.

Moving over to the headache attacks experienced by CH patients, it is remarkable that 3 out of 20 attacks (15%) are classified as *CH* with version 3 of the classification criteria in wave 1, and 14 out of 15 attacks (93.33%) in wave 2. This is a dramatic improvement in classification accuracy. Part of this can be explained by the fact that the 2 CH patients in wave 1 did not register many symptoms: 9 of the 20 attacks (45%) do not have any associated symptom, and the other 11 attacks together have only 13 associated symptoms. Both patients confirmed that they did not explicitly check all symptoms upon the registration of a headache. Analysis indeed shows that it is mostly due to the symptoms that many attacks were not classified as *CH*: 17 of the 20 attacks (85%) were unilateral around the orbit or temple, fulfilling the non-symptom-related criteria. For the second wave, the results are much better, since 14 out of the 15 attacks experienced by the 2 CH patients are classified as *CH*. Moreover, the results suggest that not taking the registered severity as a requirement for the classification is a reasonable decision: out of the 35 attacks, only 7 (20%) have a severity of "severe", and all other 28 attacks (80%) have a lower severity. As described earlier, this might be caused by a subjective lower assessment of the pain due to desensitization, as confirmed by one of the CH participants.

Finally, to come back to the *episodic TTH* classifications, this high number in CH patients (32/35, 91.43%) can also largely be explained by the lack of registered symptoms, as explained before. Only 14 of these attacks (43.75%) actually fulfill the duration criterion for *episodic TTH* (longer than 30 minutes), meaning the other 18 attacks (56.25%) are shorter than 30 minutes. From a biological viewpoint, this confirms that considering the duration for classification helps to distinguish between both, also because rapid treatment of CH attacks with oxygen or sumatriptan can result in attack abortion within minutes. Comparing this to *episodic TTH* classifications in migraine patients, the relatively high number there (57/98, 58.16%) cannot fully be explained by the lack of symptoms, since there are almost 3 symptoms on average associated to each attack. However, a possible explanation is the often-unfulfilled criterion D of the *migraine without aura* version 3 classification criteria in Table B.3, since this criterion is the logical complement of criterion D in the required set of classification criteria for *episodic TTH*.

Overall, not considering the *episodic TTH* classifications and ignoring the participant with no registered headache attacks, 12 of the 17 remaining patients (70.59%) have mostly diagnosis classifications, i.e., the disorder type for which there are the most classifications out of the patient's attacks, corresponds to the patient's diagnosis. This number is already quite good, but the absolute number of *migraine* and *CH* classifications is still rather low. Further improvement of the presented preliminary

classification system is therefore needed, by discussing and further shaping the classification criteria and the collected data that they are applied to.

A few limitations to this part of the study need to be addressed. First, we were only able to analyze a small number of participants in each group due to technical limitations of the study, i.e., limited available Empatica devices, spreading data load on server environment over time, and Empatica battery life issues. Second, also for technical reasons, the duration of the trial was only 21 days. The dynamic and cyclical nature of headache disorders often spans over multiple weeks, months or years. A three-week period therefore does not seem enough to investigate a sufficient number of attacks to investigate their complexity and to develop personalized models for individual patients. It is the authors' belief that a follow-up study should look into a minimum of three months of trial duration. Third, the participants were not asked to classify their attacks as either migraine without aura, CH or TTH. This was mainly because the objective of the set-up is to reflect the clinical reality as close as possible, by collecting clearly defined headache features by the subjects only and not self-diagnosing. We analyzed that subjects should not be qualified at this moment to provide a "ground truth" diagnosis of their individual attacks, because participants were not medically trained people and were not trained on the ICHD-3 criteria.

Moving forward in the development of an autonomous classification system for individual headache attacks, a few suggestions for improved systems can be derived from these results. First, more important than duration or treatment status to classify attacks are the symptoms of the attack which are a direct consequence of the underlying biological processes of each disorder. Second, because headache attacks within a headache syndrome can be heterogeneous intra-individually, the inclusion of new categories *probable migraine without aura* and *probable cluster headache* based on ICHD-3 guidance would be helpful to provide the clinician and patient a more nuanced and detailed overview of the different attacks. Third, because of the evolution of digital tools in medicine, it is our belief that there is a need for expert consensus within the international headache criteria to define specific criteria for different phenotypical types of headache attacks in concordance with the underlying biology of the disorders, to support longitudinal, momentary assessments in headache medicine for both clinicians and researchers. Such criteria should also consider having separate criteria based on the treatment criterion which currently limits the number of attacks for classification as shown in our results.

An important path to follow in the future improvement of the classification criteria, is the investigation of the possible inclusion of the contextual data that is collected during the mBrain study. Up to now, the classification is purely based on the static information that is entered by the patient when registering a headache attack in the mBrain app. However, much more data is available to use for classification. A potentially important source are the activity, stress and sleep events that are continuously generated by the data-driven ML algorithms, based on the physiological data

collected with the Empatica E4 wearable. From this and other data, some symptoms or triggers could be measured and validated. One example may be activity or movement measurements derived from wearable data, since CH attacks are mostly characterized by restlessness and agitation while migraine patients tend to withdraw from activities before and during attacks [6, 7]. Hence, it will be important to research, in close collaboration with headache experts, whether such contextual data can improve the classification results.

B.4.2 Data collection & interaction with automatically added events

Given the objectives of the mBrain study, the collected data should be as complete and as accurate as possible. Therefore, different parameters are important: the amount of data that is being collected, the interaction rate of patients with the applications determining the amount of feedback, and the accuracy of the data-driven ML algorithms. These aspects are analyzed and discussed below based on the general statistics of the data collection presented in Table B.8. An important aspect of this analysis is the impact of the changes implemented into the second wave as explained in Section B.2.3.2.

Amount of data collected The amount of time that the participants collected data with their Empatica device was smaller than expected during the first wave, with a little over 9 hours on average, even though participants are requested to wear the Empatica during both day and night. There are large differences between the participants, from more than 15 hours to only 5 hours of data on average per day. Different reasons were given by the participants: some found it difficult to integrate the procedure into their daily routine, while others struggled to keep their smartphone closeby (i.e., to their Empatica) all the time, causing the device to frequently disconnect. A third reason was difficulty dealing with the – sometimes shorter than expected – battery lifetime, requiring the patient to charge the battery multiple times per day. Unfortunately, these reasons did not allow for straightforward adaptations to the data collection process. Ideally, an automatic reconnection mechanism would be available that eliminates the need to manually reconnect the Empatica every time the connection is interrupted. This is something that will be available in the next Empatica SDK, which is not scheduled for release yet by Empatica. The successor of the Empatica E4 will also allow for on-device charging without interrupting the Bluetooth connection, which would also decrease the impact of battery issues. For now, it can be observed from the results of the second wave that the average connected time did increase with over 3.5 hours only thanks to stressing the importance of collecting more data during the intake visit. This is already better, but still leaves room for improvement.

Moreover, the impact of adapting the configuration of the OwnTracks app as described in Section B.2.3.4, is also visible in the results: there is an increase of more than

18 location points on average per patient per trial day, indicating that the participants' location was followed up more closely during the second wave.

Finally, one patient did not register any headache attacks throughout the trial period. This highlights the fact that recruiting patients in their active headache period is crucial. Having no headache data about a patient means that it is not possible to find any relations, making the patient's participation less useful.

Level of interaction with automatically added events The interaction rate with the automatically added events in the timeline was quite low during the first data collection wave: on average 71% of the activity events, 88% of the sleep events and 73% of the stress periods were never interacted with. However, it is important to have explicit feedback about this contextual information. To this end, as many events as possible should either be confirmed, corrected or removed.

Two main reasons for this were identified during the outtake visits: recall bias, and the large number of events per day in combination with a lack of time to process them all. Therefore, the decision was made to implement changes to the timeline in mBrain v2: a default normal timeline view with a significant reduction in number of activity events by merging sedentary activities, a restriction on the number of stress events per hour and day, and a notification about each stress event to make the patient interact with it as soon as possible and reduce possible recall bias. Moreover, during the second data collection wave, participants were stressed even more to interact and provide explicit feedback as often as possible.

The impact of the adaptations to the timeline of mBrain v2 is clearly visible in the results. The absolute average percentage of automatically added activities that were ignored decreased with more than 45%, while for stress events this decrease was more than 49%. Also for sleep events, there was an absolute decrease of more than 28%. These results demonstrate the relevance and importance of further improving these and other features that might influence the amount and accuracy of the feedback received from the participants.

Moreover, the confirmation and removal of stress events is another example of where the balance between daily life intrusion and information explicitness needs to be found. If an automatically added stress event was removed during the first wave, it was unclear why. This is in contrast to activities or sleep, where the type of activity can always be corrected, allowing for more explicit feedback. Similarly, for confirmed stress events, it was never known what the stress intensity was, since participants were not forced to edit the event and enter this intensity. Because this information is important for the further improvement of the data-driven stress detection algorithm, small additional questions were asked in mBrain v2 upon confirming or removing a stress event. In other words, the requested effort of the patient was slightly increased, in return for some more explicit feedback. Looking at the results of the second wave, the number of confirmations did not drop because of the additionally requested input,

but instead largely increased from less than to 2% to almost 45%. The number of deletions of stress events did decrease with more than 2% on average in wave 2 compared to wave 1, but the number of corrections to stress events with level 0 increased with on average more than 11%. As such, it is difficult to assess the correlation between the newly requested input and these changes in the numbers.

Outcome of interactions with automatically added events To assess how well the data-driven ML algorithms can map the patient's activities, stress and sleeping behavior, the automatically added timeline events that the patients *have* interacted with are analyzed, especially for the results of the second data collection wave. In this wave, many improvements were introduced, which positively influenced the amount of feedback received.

For the activities that were interacted with during the second data collection wave (on average 74%), on average almost 46% of the predicted activities were fully confirmed, i.e., with an explicitly confirmed type. In addition, 12% on average were confirmed as sedentary in the new normal timeline view. Less than 6% on average were corrected, and the remaining 10% were removed. First, this shows that the predictions of the activity recognition algorithms are correct in most times. Second, these results show the benefit of splitting up the timeline in two views, allowing for fine-grained or coarse-grained feedback depending on the available time of the participant. Especially since on average 88% of activities were of a sedentary type (in terms of number of events, not considering duration), the number of events in the normal timeline view was significantly reduced by merging them where possible.

For the stress events that were interacted with during the second data collection wave (on average 76%), approximately 59% of them (45% of all stress events) were confirmed with a moderate or high stress intensity, while the others were either removed or corrected to an event with stress intensity 0, which are semantically equivalent. These results are already way better than during the first wave, where only 4% of the stress events were confirmed, but there is still room for improvement.

Finally, for sleep events, approximately one third of the interactions on average was a confirmation. However, patients seemed to register their sleep periods more manually compared to activity or stress events. This could possibly be explained by two reasons. First, the sleep algorithm runs only once every 24 hours, causing these events to not be present yet in the timeline in the mornings. Second, the sleep algorithm requires physiological data from the Empatica throughout the full sleeping period to detect it. Given the average amount of Empatica data per day, some sleeping periods might therefore have not been detected.

In conclusion, it is clear that the algorithms are already able to map the patient's activities, sleep and stress reasonably well, but that further improvement of them will remain crucial. In a next phase, an interesting path to investigate is the personalization of the individual predictive models, per patient.

Translating the mBrain set-up into the real world would decrease the expected user burden because of several reasons. First, interacting with the events predicted by the ML models would no longer be required as they should be accurate enough. An easy headache attack registration process, e.g., by hitting the event button of the Empatica, would lead to a decreased registration burden. Moreover, on-device charging and automatic BLE reconnection with temporary buffering are expected future features of new Empatica devices and their SDKs, which will lead to a higher amount of collected wearable data. Finally, it should be noted that the machine-learning algorithms are generic and device-independent, meaning the Empatica could easily be replaced by another wearable that measures the same physiological data.

B.4.3 Knowledge-based detection of headache triggers

The evaluation with the example cases, demonstrated in Section B.3.3, shows that at least for physical exercise and stress, the indication by a patient of specific triggers for an attack, can be observed from these contextual events in some cases. In this evaluation, the system uses the triggers indicated by a patient to retrospectively check the data collected in the period before that headache. However, for a trigger detection system to work, triggers need to be known upfront. This is not unrealistic. If a certain event is a trigger for a headache attack, it is not unlikely that it will be a trigger for future attacks as well. During the intake visit of patients, the physician-researcher could therefore integrate questions specifically targeted at querying frequently occurring triggers. Moreover, by investing in the data-driven learning of triggers for patients based on the collected data, new triggers could be discovered, potentially including triggers that the patient is not (yet) aware of himself. In the latter case, sending a trigger alarm could be especially relevant.

For the concrete design of the individual trigger detection queries in such a system, more research is needed concerning how to define triggers, how to detect each trigger based on the available data, the optimal personalized time window, among other things. Also, collecting other contextual data could enlarge the set of detectable headache triggers. An example could be the detection of flickering light or loud noise through the collection of light intensity and noise data.

While researching those new systems, it should not be forgotten, however, that these "triggers" may also be a misconception of the presence of premonitory symptoms already happening before the trigger and headache attack occur [61]. For example, chocolate may not be a trigger for migraine but rather the craving towards sweets may be a premonitory symptom already present before the patient eats chocolate [70].

In summary, the fact that currently indicated triggers can often be backed up with the collected data, proves the potential usefulness of a trigger detection system. In addition, it is another example of why it is important and useful to invest time and resources into the collection of a wide range of physiological and contextual data

through the Empatica E4 wearable and the various applications, and the design of data-driven algorithms that analyze this data to detect certain events, in order to improve the continuous follow-up of headache patients.

B.5 Conclusions

In this appendix, the set-up and first results of the mBrain study are discussed. mBrain is an exploratory, observational research study that investigates how to move from the intermittent, subjective follow-up and classification of headaches based on self-reported data, towards a more continuous, semi-autonomous, objective follow-up and classification that is based on a combination of self-reported data, and objective physiological and contextual data. Therefore, physiological data is automatically collected with the Empatica E4 wearable. Data-driven ML algorithms use this data to detect the activities, stress events and sleeping behavior of the patients. Using a mobile application, patients can interact with these events, and keep a diary of other contextual and headache-specific data.

As a first subquestion, the study has investigated how to collect as much objective and explicit data as possible about a patient's headache attacks and relevant context. After a first data collection wave, several changes implemented into the set-up have successfully improved the level and accuracy of the received feedback on predictions of the ML algorithms during a second data collection wave. This shows that it is relevant to keep further improving and fine-tuning this set-up, while balancing between daily life intrusion and information explicitness, to obtain a complete and correct view on the patient's context and lifestyle.

Second, the study has researched how to design an autonomous classification system for individual headache attacks. Therefore, a knowledge-based system was designed to classify registered attacks as either migraine without aura, CH, or episodic TTH. Different versions of classification criteria were designed, starting from the ICHD-3 diagnostic criteria. The results show that strictly applying the ICHD-3 criteria on individual attacks does not yield good classification results. Adapted versions yield better results, leading to mostly diagnosis classifications for 12 of the 18 patients if episodic TTH classifications are ignored. However, the absolute number of migraine without aura (28/98) and CH classifications (17/35) is still rather low. Therefore, further shaping the classification criteria and data they are applied to is required. An interesting path to investigate here is whether and how the events detected by the ML algorithms can be integrated into the classification process. Moreover, specifically for migraine patients, it should be further researched how to deal with the intra-individual heterogeneity of migraine attacks.

Third, to integrate the output of the data-driven ML algorithms for the continuous follow-up and classification of headache attacks, it should present an accurate view on the patients' context. The results of the second data collection wave show that

this is largely true for activity events, and that serious improvements have been made for stress and sleep events. Therefore, further refinement of the different algorithms will remain important. It should be investigated whether the personalization of the individual predictive models can increase the overall accuracy.

Fourth and final, the study has taken the first steps to investigate how the physiological, contextual and headache-related data of patients can be linked to be valuable for the continuous follow-up of headaches. To this end, two example cases have demonstrated the potential of using the outputs of the data-driven ML algorithms for the knowledge-based detection of known headache triggers. In addition, it will be useful to research how headache triggers for specific patients can be discovered by data-driven learning techniques. In summary, this highlights the potential of focusing on hybrid AI for the future improvement of continuous headache follow-up, classification and trigger detection.

Funding

This work was partially funded by imec via the AAA Context-aware Health Monitoring project. N.V. received funding from Ghent University Hospital for his research (Fonds voor Innovatie en Klinisch Onderzoek, 2019). B.S. (1SA0219N) is funded by a strategic base research grant of Fonds Wetenschappelijk Onderzoek (FWO), Belgium. J.V.D.D. is funded by a doctoral fellowship of Ghent University (BOF).

Acknowledgments

The authors thank Bartosz Gajda for laying the basics of the mobile mBrain app and the back-end of the system. Moreover, the authors thank Olivier Janssens for setting the first steps in designing the stress and activity detection algorithms. Finally, the authors would like to thank all study subjects for their collaboration and efforts in the mBrain study.

Availability of data and materials

The datasets generated during and/or analyzed during the current study are not publicly available, as imposed by the Ethics Committee of the Ghent University Hospital due to the protection of privacy of the participants. The DAHCC ontology is publicly available at https://github.com/IBCNServices/DAHCC-ontology. The version of the DAHCC ontology used in this appendix is the version at https://github.com/IBCNServices/DAHCC-ontology/tree/b83abcd90ffcb8faf9c2a4c8822e124a58e1ed63.

Ethics approval and consent to participate

The protocol of the mBrain study has been approved by the Ethics Committee of the Ghent University Hospital, under the name of the "COPIMAC study" (BC-07403). Written Informed Consent for participation to the study was obtained from all participants of the mBrain study, as part of the protocol approved by the Ethics Committee of the Ghent University Hospital. The methods in the protocol are in accordance with all relevant guidelines and regulations.

References

[1] L. Stovner, K. Hagen, R. Jensen, Z. Katsarava, R. Lipton, A. Scher, T. Steiner, and J. Zwart. *The global burden of headache: a documentation of headache prevalence and disability worldwide*. Cephalalgia, 27(3):193–210, 2007. doi:10.1111/j.1468-2982.2007.01288.x.

[2] World Health Organization. *Headache disorder – fact sheet*, 2016. Accessed: 2020-10-23. Available from: https://www.who.int/news-room/fact-sheets/detail/headache-disorders.

[3] Headache Classification Committee of the International Headache Society (IHS). *The International Classification of Headache Disorders, 3rd edition*. Cephalalgia, 38(1):1–211, 2018. doi:10.1177/0333102413485658.

[4] P. J. Goadsby. *Chapter 13: Headache*. In J. L. Jameson, A. S. Fauci, D. L. Kasper, S. L. Hauser, D. L. Longo, and J. Loscalzo, editors, Harrison's Principles of Internal Medicine. McGraw-Hill, 20th edition, 2018. Available from: https://accesspharmacy.mhmedical.com/content.aspx?bookid=2129&sectionid=192011003.

[5] L. J. Stovner, E. Nichols, T. J. Steiner, F. Abd-Allah, A. Abdelalim, R. M. Al-Raddadi, M. G. Ansha, A. Barac, I. M. Bensenor, L. P. Doan, et al. *Global, regional, and national burden of migraine and tension-type headache, 1990–2016: a systematic analysis for the Global Burden of Disease Study 2016*. The Lancet Neurology, 17(11):954–976, 2018. doi:10.1016/S1474-4422(18)30322-3.

[6] P. J. Goadsby, P. R. Holland, M. Martins-Oliveira, J. Hoffmann, C. Schankin, and S. Akerman. *Pathophysiology of Migraine: A Disorder of Sensory Processing*. Physiological Reviews, 2017. doi:10.1152/physrev.00034.2015.

[7] J. Hoffmann and A. May. *Diagnosis, pathophysiology, and management of cluster headache*. The Lancet Neurology, 17(1):75–83, 2018. doi:10.1016/S1474-4422(17)30405-2.

[8] A. Snoer, N. Lund, R. Beske, A. Hagedorn, R. H. Jensen, and M. Barloese. *Cluster headache beyond the pain phase: a prospective study of 500 attacks*. Neurology, 91(9):e822–e831, 2018. doi:10.1212/01.wnl.0000542491.92981.03.

[9] Z. Katsarava, M. Mania, C. Lampl, J. Herberhold, and T. J. Steiner. *Poor medical care for people with migraine in Europe – evidence from the Eurolight study*. The Journal of Headache and Pain, 19(1), 2018. doi:10.1186/s10194-018-0839-1.

[10] P. Vo, N. Paris, A. Bilitou, T. Valena, J. Fang, C. Naujoks, A. Cameron, F. d. R. de Vulpillieres, and F. Cadiou. *Burden of migraine in Europe using self-reported digital diary data from the Migraine Buddy© application*. Neurology and Therapy, 7(2):321–332, 2018. doi:10.1007/s40120-018-0113-0.

[11] R. Aggarwal, S. Ringold, D. Khanna, T. Neogi, S. R. Johnson, A. Miller, H. I. Brunner, R. Ogawa, D. Felson, A. Ogdie, D. Aletaha, and B. M. Feldman. *Distinctions Between Diagnostic and Classification Criteria?* Arthritis Care & Research, 67(7):891–897, 2015. doi:10.1002/acr.22583.

[12] A. Roesch, M. A. Dahlem, L. Neeb, and T. Kurth. *Validation of an algorithm for automated classification of migraine and tension-type headache attacks in an electronic headache diary*. The Journal of Headache and Pain, 21(1), 2020. doi:10.1186/s10194-020-01139-w.

[13] Healint. *Migraine Buddy*, 2020. Accessed: 2021-01-19. Available from: https://migrainebuddy.com.

[14] M. T. Minen, T. Gumpel, S. Ali, F. Sow, and K. Toy. *What are headache Smartphone application (app) users actually looking for in apps: a qualitative analysis of app reviews to determine a patient centered approach to headache Smartphone Apps*. Headache: The Journal of Head and Face Pain, 60(7):1392–1401, 2020. doi:10.1111/head.13859.

[15] G. Nappi, R. Jensen, R. E. Nappi, G. Sances, P. Torelli, and J. Olesen. *Diaries and calendars for migraine. A review*. Cephalalgia, 26(8):905–916, 2006. doi:10.1111/j.1468-2982.2006.01155.x.

[16] R. Sadovsky and D. W. Dodick. *Identifying migraine in primary care settings*. The American Journal of Medicine Supplements, 118:11–17, 2005. doi:10.1016/j.amjmed.2005.01.015.

[17] Softarch Technologies AS. *iMigraine – migraine monitor and headache tracking*, 2020. Accessed: 2021-01-19. Available from: https://play.google.com/store/apps/details?id=com.iMigraine.app.

[18] M3 Technology. *Migraine Headache Diary HeadApp*, 2019. Accessed: 2021-01-19. Available from: https://play.google.com/store/apps/details?id=it.mthree.headapp.user.android.lite.

[19] RPM Healthcare. *Migraine Monitor*, 2020. Accessed: 2021-01-19. Available from: https://play.google.com/store/apps/details?id=com.migrainemonitor.

[20] Christophe DELAGE. *My Cluster Headache*, 2016. Accessed: 2021-01-19. Available from: https://play.google.com/store/apps/details?id=eu.monavf.android.monavf.

[21] HD Yoga Poses Wallpapers. *Tension Headache*, 2017. Accessed: 2021-01-19. Available from: https://play.google.com/store/apps/details?id=com.Tension.Headache.

[22] AR Productions Inc. *Headache Log*, 2020. Accessed: 2021-01-19. Available from: https://play.google.com/store/apps/details?id=arproductions.andrew.headachelog.

[23] C. Tassorelli, R. Jensen, M. Allena, R. De Icco, Z. Katsarava, J. Miguel Lainez, J. A. Leston, R. Fadic, S. Spadafora, M. Pagani, et al. *The added value of an electronic monitoring and alerting system in the management of medication-overuse headache: A controlled multicentre study*. Cephalalgia, 37(12):1115–1125, 2017. doi:10.1177/0333102416660549.

[24] D. L. van de Graaf, G. G. Schoonman, M. Habibović, and S. C. Pauws. *Towards eHealth to support the health journey of headache patients: a scoping review*. Journal of Neurology, 268:3646–3665, 2020. doi:10.1007/s00415-020-09981-3.

[25] M. T. Minen, S. Adhikari, E. K. Seng, T. Berk, S. Jinich, S. W. Powers, and R. B. Lipton. *Smartphone-based migraine behavioral therapy: a single-arm study with assessment of mental health predictors*. npj Digital Medicine, 2(1), 2019. doi:10.1038/s41746-019-0116-y.

[26] M. Mosadeghi-Nik, M. S. Askari, and F. Fatehi. *Mobile health (mHealth) for headache disorders: A review of the evidence base*. Journal of Telemedicine and Telecare, 22(8):472–477, 2016. doi:10.1177/1357633X16673275.

[27] A. S. Hundert, A. Huguet, P. J. McGrath, J. N. Stinson, and M. Wheaton. *Commercially available mobile phone headache diary apps: a systematic review*. JMIR mHealth and uHealth, 2(3), 2014. doi:10.2196/mhealth.3452.

[28] P. Kropp, B. Meyer, W. Meyer, and T. Dresler. *An update on behavioral treatments in migraine–current knowledge and future options*. Expert Review of Neurotherapeutics, 17(11):1059–1068, 2017. doi:10.1080/14737175.2017.1377611.

[29] M. T. Minen, E. J. Stieglitz, R. Sciortino, and J. Torous. *Privacy issues in smartphone applications: an analysis of headache/migraine applications*. Headache: The Journal of Head and Face Pain, 58(7):1014–1027, 2018. doi:10.1111/head.13341.

[30] R. Potter, K. Probyn, C. Bernstein, T. Pincus, M. Underwood, and M. Matharu. *Diagnostic and classification tools for chronic headache disorders: a systematic review*. Cephalalgia, 39(6):761–784, 2019. doi:10.1177/0333102418806864.

[31] D. Phillip, A. Lyngberg, and R. Jensen. *Assessment of headache diagnosis. A comparative population study of a clinical interview with a diagnostic headache diary*. Cephalalgia, 27(1):1–8, 2007. doi:10.1111/j.1468-2982.2007.01239.x.

[32] H. C. Diener, M. Ashina, I. Durand-Zaleski, T. Kurth, M. Lantéri-Minet, R. B. Lipton, D. A. Ollendorf, P. Pozo-Rosich, C. Tassorelli, and G. Terwindt.

Health technology assessment for the acute and preventive treatment of migraine: A position statement of the International Headache Society. Cephalalgia, 41(3):279–293, 2021. doi:10.1177/0333102421989247.

[33] Empatica. *E4 wristband*, 2020. Accessed: 2020-10-23. Available from: https://www.empatica.com/research/e4.

[34] R. J. Cole, D. F. Kripke, W. Gruen, D. J. Mullaney, and J. C. Gillin. *Automatic sleep/wake identification from wrist activity*. Sleep, 15(5):461–469, 1992. doi:10.1093/sleep/15.5.461.

[35] M. Gjoreski, M. Luštrek, M. Gams, and H. Gjoreski. *Monitoring stress with a wrist device using context*. Journal of Biomedical Informatics, 73:159–170, 2017. doi:10.1016/j.jbi.2017.08.006.

[36] P. Schmidt, A. Reiss, R. Duerichen, C. Marberger, and K. Van Laerhoven. *Introducing WESAD, a multimodal dataset for Wearable Stress and Affect Detection*. In ICMI '18: Proceedings of the 20th ACM International Conference on Multimodal Interaction, 2018. doi:10.1145/3242969.3242985.

[37] M. Stojchevska, B. Steenwinckel, J. Van Der Donckt, M. De Brouwer, A. Goris, F. De Turck, S. Van Hoecke, and F. Ongenae. *Assessing the added value of context during stress detection from wearable data*. BMC Medical Informatics and Decision Making, 22(1), 2022. doi:10.1186/s12911-022-02010-5.

[38] OwnTracks.org. *OwnTracks – Your location companion*, 2020. Accessed: 2020-10-23. Available from: https://owntracks.org/.

[39] R. B. Lipton, D. C. Buse, C. B. Hall, H. Tennen, T. A. DeFreitas, T. M. Borkowski, B. M. Grosberg, and S. R. Haut. *Reduction in perceived stress as a migraine trigger: testing the "let-down headache" hypothesis*. Neurology, 82(16):1395–1401, 2014. doi:10.1212/WNL.0000000000000332.

[40] R. B. Lipton, R. Cady, W. Stewart, K. Wilks, and C. Hall. *Diagnostic lessons from the spectrum study*. Neurology, 58(9 suppl 6):S27–S31, 2002. doi:10.1212/wnl.58.9_suppl_6.s27.

[41] W. F. Stewart, R. B. Lipton, K. B. Kolodner, J. Sawyer, C. Lee, and J. N. Liberman. *Validity of the Migraine Disability Assessment (MIDAS) score in comparison to a diary-based measure in a population sample of migraine sufferers*. Pain, 88(1):41–52, 2000. doi:10.1016/S0304-3959(00)00305-5.

[42] W. F. Stewart, R. B. Lipton, J. Whyte, A. Dowson, K. Kolodner, J. a. Liberman, and J. Sawyer. *An international study to assess reliability of the Migraine Disability Assessment (MIDAS) score*. Neurology, 53(5):988–988, 1999. doi:10.1212/wnl.53.5.988.

[43] W. F. Stewart, R. Lipton, K. Kolodner, J. Liberman, and J. Sawyer. *Reliability of the migraine disability assessment score in a population-based sample of headache sufferers*. Cephalalgia, 19(2):107–114, 1999. doi:10.1046/j.1468-2982.1999.019002107.x.

[44] G. I. Kempen. *The MOS Short-Form General Health Survey: single item vs multiple measures of health-related quality of life: some nuances*. Psychological Reports, 70(2):608–610, 1992. doi:10.2466/pr0.1992.70.2.608.

[45] G. Kempen, E. Brilman, J. Heyink, and J. Ormel. *Het meten van de algemene gezondheidstoestand met de MOS Short-Form General Health Survey (SF-20): een handleiding*. Noordelijk Centrum voor Gezondheidsvraagstukken, Rijksuniversiteit Groningen, 1995.

[46] B. C. Martin, D. S. Pathak, M. I. Sharfman, J. U. Adelman, F. Taylor, W. J. Kwong, and P. Jhingran. *Validity and reliability of the migraine-specific quality of life questionnaire (MSQ Version 2.1)*. Headache: The Journal of Head and Face Pain, 40(3):204–216, 2001. doi:10.1046/j.1526-4610.2000.00030.x.

[47] IDLab (Ghent University – imec). *Obelisk*, 2020. Accessed: 2020-10-23. Available from: https://obelisk.ilabt.imec.be/api/v2/docs/.

[48] K. Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc., 2013.

[49] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and Running: Dive into the future of infrastructure*. O'Reilly Media, Inc., 2017.

[50] Ghent University – imec. *iLab.t Virtual Wall*, 2020. Accessed: 2020-10-23. Available from: https://doc.ilabt.imec.be/ilabt/virtualwall.

[51] D. Merkel. *Docker: lightweight linux containers for consistent development and deployment*. Linux Journal, 2014(239), 2014. Available from: https://dl.acm.org/doi/10.5555/2600239.2600241.

[52] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. *Semantics for the Internet of Things: Early Progress and Back to the Future*. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1):1–21, 2012. doi:10.4018/jswis.2012010101.

[53] T. R. Gruber. *A translation approach to portable ontology specifications*. Knowledge Acquisition, 5(2):199–220, 1993. doi:10.1006/knac.1993.1008.

[54] B. Steenwinckel, M. De Brouwer, M. Stojchevska, J. Van Der Donckt, J. Nelis, J. Ruyssinck, J. van der Herten, K. Casier, J. Van Ooteghem, P. Crombez, F. De Turck, S. Van Hoecke, and F. Ongenae. *DAHCC: Data Analytics For Health and Connected Care: Connecting Data Analytics to Healthcare Knowledge in an IoT environment*, 2022. Accessed: 2022-02-03. Available from: https://dahcc.idlab.ugent.be.

[55] M. Girod-Genet, L. N. Ismail, M. Lefrançois, and J. Moreira. *ETSI TS 103 410-8 V1.1.1 (2020-07): SmartM2M; Extension to SAREF; Part 8: eHealth/Ageing-well Domain*. Technical report, ETSI Technical Committee Smart Machine-to-Machine communications (SmartM2M), 2020. Available from: https://www.etsi.org/deliver/etsi_ts/103400_103499/10341008/01.01.01_60/ts_10341008v010101p.pdf.

[56] M. Compton, P. Barnaghi, L. Bermudez, R. García-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. Le Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor. *The SSN ontology of the W3C semantic sensor network incubator group*. Journal of Web Semantics, 17:25–32, 2012. doi:10.1016/j.websem.2012.05.003.

[57] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride. *RDF 1.1 concepts and abstract syntax*. W3C Recommendation, World Wide Web Consortium (W3C), 2014. Available from: https://www.w3.org/TR/rdf11-concepts/.

[58] S. Harris and A. Seaborne. *SPARQL 1.1 Query Language*. W3C Recommendation, World Wide Web Consortium (W3C), 2013. Available from: https://www.w3.org/TR/sparql11-query/.

[59] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. *RDFox: A highly-scalable RDF store*. In The Semantic Web - ISWC 2015: Proceedings, Part II of the 14th International Semantic Web Conference, pages 3–20, Cham, Switzerland, 2015. Springer. doi:10.1007/978-3-319-25010-6_1.

[60] M. J. Marmura. *Triggers, protectors, and predictors in episodic migraine*. Current Pain and Headache Reports, 22(12):1–9, 2018. doi:10.1007/s11916-018-0734-0.

[61] N. Karsan and P. J. Goadsby. *Biological insights from the premonitory symptoms of migraine*. Nature Reviews Neurology, 14(12):699–710, 2018. doi:10.1038/s41582-018-0098-4.

[62] L. Kelman. *The triggers or precipitants of the acute migraine attack*. Cephalalgia, 27(5):394–402, 2007. doi:10.1111/j.1468-2982.2007.01303.x.

[63] D. Andress-Rothrock, W. King, and J. Rothrock. *An analysis of migraine triggers in a clinic-based population*. Headache: The Journal of Head and Face Pain, 50(8):1366–1370, 2010. doi:10.1111/j.1526-4610.2010.01753.x.

[64] A. Hagedorn, A. Snoer, R. Jensen, B. Haddock, and M. Barloese. *The spectrum of cluster headache: A case report of 4600 attacks*. Cephalalgia, 39(9):1134–1142, 2019. doi:10.1177/0333102419833081.

[65] D. Russell. *Cluster headache: severity and temporal profiles of attacks and patient activity prior to and during attacks*. Cephalalgia, 1(4):209–216, 1981. doi:10.1046/j.1468-2982.1981.0104209.x.

[66] R. G. Kaniecki. *Migraine and tension-type headache: an assessment of challenges in diagnosis*. Neurology, 58(9 suppl 6):S15–S20, 2002. doi:10.1212/wnl.58.9_suppl_6.s15.

[67] R. Peatfield. *Migrainous features in cluster headache*. Current Pain and Headache Reports, 5(1):67–70, 2001. doi:10.1007/s11916-001-0012-3.

[68] T.-H. Lai, J.-L. Fuh, and S.-J. Wang. *Cranial autonomic symptoms in migraine: characteristics and comparison with cluster headache*. Journal of Neurology, Neurosurgery & Psychiatry, 80(10):1116–1119, 2009. doi:10.1136/jnnp.2008.157743.

[69] A. L. Vollesen, S. Benemei, F. Cortese, A. Labastida-Ramírez, F. Marchese, L. Pellesi, M. Romoli, M. Ashina, and C. Lampl. *Migraine and cluster headache – the common link*. The Journal of Headache and Pain, 19(1), 2018. doi:10.1186/s10194-018-0909-4.

[70] M. Nowaczewska, M. Wiciński, W. Kaźmierczak, and H. Kaźmierczak. *To Eat or Not to eat: A Review of the Relationship between Chocolate and Migraines*. Nutrients, 12(3):608, 2020. doi:10.3390/nu12030608.