Generating Multiple Conceptual Models from Behavior-Driven Development Scenarios

Abstract

Researchers have proposed that generating conceptual models automatically from user stories might be useful for agile software development. It is, however, unclear from the state-of-the-art *what* a consistent and complementary set of models to generate is, *how* these models can be generated such that relationships and dependencies in a set of related user stories are unveiled, and *why* these models are useful in agile software development projects. In this paper, we address these questions through a Design Science research study. First, we define four stylized versions of Unified Modeling Language (UML) diagrams (i.e., use case diagram, class diagram, activity diagram, state machine diagram) that will be the target of the model generation. Although these stylized UML diagrams have a reduced abstract syntax, they offer different perspectives on the software system in focus with potential usefulness for requirements and software engineering. Second, we develop an automated model generation approach based on different design artefacts including a Natural Language Processing (NLP) tool that implements our approach. Key to our solution is the use of the Behavior-Driven Development (BDD) scenario template to document user stories. Using an example set of BDD scenarios as source of the model generation, we demonstrate the feasibility of our approach via the NLP tool that implements our approach. Third, we conduct an empirical study with experts in agile software development involving the researcher-guided interactive use of our tool to explore the use of the generated models. This study shows the perceived usefulness of the models that our tool can generate and identifies different uses and benefits of the models for requirements analysis, system design, software implementation, and testing in projects that employ agile methods.

Keywords: conceptual modeling, user stories, agile software development, behavior-driven development, automated model generation

1 Introduction

In Agile software development, requirements documentation is mainly limited to user stories [1]. A user story is a simple description of a feature of the working software as it is expected by a user [2, 3]. Because of the substantial number of user stories that are written in Agile software development projects (e.g., in SAFe projects that involve multiple teams [4]), the project team may encounter difficulties in maintaining, tracing, and managing user stories [5]. While the focus of requirements engineering in Agile development is on facilitating the transfer of ideas from the customer to the development team [5], it has been shown that a deep understanding of the domain and sharing the domain knowledge are crucial factors for the success of software development projects [6]. Considering that user stories might be the only documentation available to the project team, acquiring an overall understanding of the system's required features and their dependencies is challenging.

The further specification of requirements described by user stories is usually done using Behavior-Driven Development (BDD) scenarios. A BDD scenario describes how the system behaves in response to what is required as described in a user story [7]. BDD scenarios and their associated user stories contain domain information and business logic that drives the design of the system's architecture. Recently, research on Agile software development is addressing the challenge of how to make user stories 'executable' to more efficiently, and ideally automatically, capture that knowledge to generate working software [8, 9].

This paper posits that the use of conceptual models generated from BDD scenarios helps addressing such challenges related to requirements analysis and software design in Agile software development. In software engineering, conceptual models support requirements analysis and system design activities [10]. Conceptual models are graphical representations of a domain to be supported by a system that help acquiring and sharing the understanding of the requirements and maintaining an overview [11, 12]. Furthermore, they are required in model-driven development approaches that aim at reducing the time and cost needed to produce software [13].

It is well understood that, despite not being promoted by the Agile paradigm, the use of conceptual models is beneficial for requirements engineering, system design, and software testing in Agile software development [14]. For instance, Trkman et al. [15] suggest that process models should be used to better understand the dependencies among user stories. Wautelet et al. [16] suggest a process for transforming sets of user stories into use case models that allow for a visual representation of the system functions. Jacqueira et al. [17] provide heuristics to map user stories into goal models that provide additional documentation for the project team. Elallaoui et al. [18] develop sequence diagrams from user stories to automatically generate test cases. Gilson et al.[19] create robustness diagrams that visualize use case scenarios to analyse how user stories impact design decisions during sprint planning and implementation. These referenced papers are just examples. A recent systematic literature review by Raharjana et al. [20] established the Natural Language Processing (NLP) based generation of conceptual models (e.g. [21]) and other software engineering artifacts from user stories as a distinct research topic.

Reviewing the state-of-the-art in creating conceptual models from user stories, we identify three research gaps. First, apart from Wautelet et al. [22], studies focus on generating just one kind of conceptual model (e.g., use case model [18], [23], [24]; class diagram [25], [26]; goal model [27], [28]; OWL model [29]) whereas in software development different conceptual models are used to cover different perspectives (e.g., data, process, function, interaction) of the software system in development. Second, models are generated from user stories for a

particular use (e.g., generating test cases [18], reasoning over requirements [30], obtaining a domain model [29]) or the intended use is implicit (e.g., [31]). There is a need to explore more deeply the different ways that models generated from user stories could be useful in Agile software development. Third, the information on the user stories that is the input to the model generation algorithms is not sufficient to identify dependencies between user stories. For instance, despite the usefulness of process models shown by Trkman et al. [15], none of the reviewed studies aims to generate a process model that shows how the system functionality described by a set of related user stories is embedded into the logic of a process that is to be performed. Similarly, there are no approaches to create models that show how the informational objects maintained by a system change state through the actions described by related user stories.

These three research gaps lead us to the following questions that guide our research.

RQ1: what conceptual models rich in information about the user stories and their dependencies are useful for Agile software development?

RQ2: how to generate these conceptual models?

RQ3: why are the generated models useful, or stated differently, what is their use?

Based on these questions, we formulate our research objectives. First, we aim at defining a set of conceptual models for requirements analysis and system design to be generated from user stories, that is consistent, meaning that the information in the different models is not contradictory, and that is complementary, meaning that the models offer different perspectives on the software system and do not overlap too much in information content. To ensure this complementarity and consistency, we define these models as 'stylized' versions of UML diagrams such that all models use the same standard modeling language for software engineering (i.e., UML) and resemble standard UML diagrams as close as possible based on the information available in the user stories (i.e., 'stylized'). Second, we develop a technique to generate these models. Novel in our approach is that, to capture information on dependencies between user stories, we do not generate the models directly from the user stories but from BDD scenarios, which not only specify user stories but also their precondition, trigger, and postcondition. Third, we show how these models are useful, though we restrict this demonstration to selected requirements validation and analysis use cases as showing this for all possible use cases requires a study on its own which does not fit the scope of this paper but could be taken up in future research. Instead of systematically exploring all possible uses of the models that we generate from BDD scenarios, we present in this paper a first empirical validation by assessing the usefulness of the generated models as perceived by experts in Agile software development.

For our research, we follow the Design Science paradigm [32, 33] to achieve our objectives and accordingly we develop several artifacts. A first artifact is a theoretical framework that allows interpreting dependencies between user stories and provides a uniform representation format for a set of BDD scenarios that document related user stories (e.g., a theme or epic as used in the Scrum or SAFe methodologies) to be used as input for the model generation. A second artifact is a set of mappings, operationalized as mapping rules that can be automated, between the concepts of BDD scenarios and the concepts of the stylized UML diagrams that we wish to generate. A third artifact are the algorithms that implement the mapping rules and are coded in an executable software program that we use as research tool for investigating the usefulness of the generated models.

The paper is structured as follows. Section 2 presents the background of the paper. The Connextra template for writing user stories and the Gherkin format for Behavior-Driven

Development (BDD) scenarios are introduced here. Section 3 presents our research methods and process which is guided by the Design Science. Section 4 presents the selection of conceptual models that we generate from a set of related user stories. Section 5 presents an agent-based framework that we use to interpret user stories and their dependencies based on the information in the BDD scenarios. This section also presents the structured representation format that we use for mapping the information found in a set of BDD scenarios that document related user stories to the concepts of the agent-based framework. Section 6 then presents and illustrates the rules for generating from the BDD scenarios, stylized UML diagrams that correspond to the selected conceptual models. Section 7 demonstrates the application of the tool that executes the algorithms that implement the mapping rules. As a proof of concept of the usefulness of the models for Agile software development, we sketch scenarios of how the generated models can be used for validating and analyzing requirements expressed in related user stories. While these scenarios demonstrate the assumed usefulness of models generated from BDD scenarios, Section 8 presents an empirical study in which seasoned practitioners of Agile software development express their opinion on the perceived usefulness of models generated using the tool. Section 9 is the discussion section that states our contribution related to the state-of-the-art and discusses limitations and future research. Finally, section 10 is the conclusion.

2 Background

2.1 User stories written using the Connextra template

User stories express three distinct aspects of the features that users expect from a software system: (1) who wants the feature; (2) what functionality the user wants the system to provide; and (3) why the users need this functionality [16] . The structure of a user story can thus be divided into three parts – role, means, and ends, which map to the interrogatives of who, what, and why respectively.

In a survey on the use of user stories by practitioners, Lucassen et al. [34] found that around 60% of the respondents use the Connextra template. In the Connextra template [2] for writing user stories, each part (interrogative) has a distinctive indicator to separate that part from other parts. The role (who), with indicator “As a”, describes a user of the system. This user role can represent a person that directly interacts with the system's interfaces (e.g., an end user) or represent any other system stakeholder that requires the functionality described in the user story (e.g., a manager that requires a secretary to generate a report using the system). The role can refer to a type of user or to an instance of such type. The means (what), with indicator “I want to”, consists of an object and an action to be performed on this object (e.g., create report). The ends (why), with indicator “so that”, describes the benefits of the functionality for the user from which perspective the user story is formulated (e.g., the manager wants to be informed about something).

An example user story written in the Connextra template is:

> *As a customer, I want to cancel a service request so that the team can focus on other active requests.*

This user story could for instance be taken from a Scrum epic that describes required functionality of a software system supporting service request management (e.g., a system supporting the ITIL 4 service request management practice [35]. The user role is a customer. The user story describes that the customer expects the system to support the action of cancelling a service request (i.e., an information object to be stored and processed by the system).

Cancelling a service request is a means to achieve the desired end of freeing up resources of the team that is responsible for handling service requests.

2.2 Behavior-Driven Development scenarios written in the Gherkin language

A BDD scenario consists of a feature title, an associated user story, and the actual scenario that defines the system behavior. Using the Gherkin format, this scenario is marked by three keywords – "Given", "When", and "Then" [7], for three distinct parts of the scenario – precondition, trigger and postcondition. The "Given" indicator marks the precondition part in the scenario, in which the context is described that is assumed by the user story. The action to be performed on the object as described in the means part of the user story, can only be triggered in this context, which is expressed in terms of one or more system objects and their states, where object states can be the result of actions described in other user stories (i.e., thus indicating dependencies between user stories). The trigger part, with indicator "when", describes one or more events that trigger the action described in the user story to be performed. Finally, the "then" indicator marks the postcondition part of the scenario that describes the outcome(s) of the user story in terms of object states achieved. Within each of the parts of a scenario, the conjunction operator "and" can be used.

For the feature *service request cancellation*, a scenario for the example user story of sub-section 2.1 could be:

> *Given a service request is submitted or fixed, when the customer decides to cancel the service request, then the service request should be canceled.*

The scenario specifies that the customer can request to cancel a service request only when this service request is submitted or fixed, which are the states of the service request object in which the cancel action described by the user story is allowed. The outcome of the user story is that the service request object is in state canceled.

The template that we assume is used for writing BDD scenarios, is shown in Table 1 along with the example. Note that this scenario embeds both the Connextra template for writing user stories and the Gherkin format for writing scenarios.

BDD Scenario Template	**Example**
As a [role], *I want to [means]* *so that [ends].* *Given [context] and/or [some more context],* *when [some event occurs] and/or [some other event occurs],* *then [outcome] and [some other outcome].*	*As a customer,* *I want to cancel a service request* *so that the team can focus on other active requests.* *Given a service request is submitted or fixed,* *when the customer decides to cancel the service request,* *then the service request should be cancelled.*

Table 1. The BDD Scenario Template (based on [7])

Although templates for writing user stories are widely used [36], there are no specific guidelines on how to use them, and often user stories are not complete (e.g., missing indicators) and are not written in a single sentence [37]. We therefore make certain assumptions about the use of the BDD scenario template shown in Table 1. These assumptions were inspired by good quality principles for BDD scenarios [38].

First, we assume that the *role*, *means*, *context*, and *outcome* elements of the template are always filled. Exceptionally, the *context* is not specified but we believe this situation is rare (e.g., a user story that describes the very first action to be performed with the system).

Second, although users can perform (or have the system perform) different actions affecting different system objects, we define the granularity of a BDD scenario as restricted to a single type of user performing a single action on one specific object. Hence, we assume the combination of user, action, and object to be unique in each BDD scenario in scope (i.e., within the set of related user stories considered). If more than one type of user desires the same action to be performed on the same object, then a separate BDD scenario is written for each type of user. If more than one object or one action is involved, then it is assumed that the BDD scenario can be broken down into several BDD scenario, each modeling one action to be performed on one object. An example is a supervisor who can approve a personal leave and a sick leave for an employee. In this case, there should be two separate BDD scenario, one for approving the personal leave and the other for approving the sick leave.

Third, although many scenarios can be associated to the same user story, describing different contexts, triggers, or outcomes for the user story, we assume a one-to-one mapping of user story and scenario. In other words, we assume that each BDD scenario is composed of a unique user story and a unique scenario. For instance, the example scenario could be split into two separate scenarios with respective contexts *service request is submitted* and *service request is fixed*, however, this would result in a one-to-many mapping of user story and scenarios, and thus two BDD scenarios each with a different scenario but with the same user story. Therefore, as shown in Table 1, we accept disjunctions (i.e., "or") in the "Given" and "when" parts of the scenario. However, we do not extend the template to include a disjunction in the user story postcondition (i.e., "then") as we expect user stories to have a deterministic outcome.

3 Methodology

As achieving our research objectives requires creating different artifacts (i.e., (1) an agent-based framework to interpret user stories and dependencies between user stories based on the information in BDD scenarios; (2) mappings from a structured representation of this information to concepts of stylized UML diagrams; (3) algorithms and a tool to implement the mapping rules), our research approach follows the Design Science methodology [32]. Figure 1 shows how we applied the Design Science Research (DSR) process prescribed by [33]. The DSR process allows for iterative design of the artifacts with an explicit loop back after the 'Evaluation' and 'Communication' steps. While we applied iterative design in the 'Define objectives for a solution', 'Design and development', and 'Demonstration' steps, for which the paper presents the consolidated results, possible further design iterations after 'Evaluation' and 'Communication' are yet to be performed.

The DSR process entry point [39] for the research presented in this paper is the Define objectives for a solution step. Our actual research process started with a problem investigation employing systematic literature review and expert interviews, based on which the research gaps stated in Section 1 (Introduction) were identified. The research design, activities. and results of this 'Identify problem and motivate' DSR process step are available in a technical report that is submitted for publication.[1]

The solution objectives for the artifacts are defined by the models we wish to generate from a set of related user stories to support requirements analysis and system design activities in Agile software development. This requires an investigation of our first research objective, the results

[1] https://github.com/conceptualmodel/modelsfromuserstory/tree/paper-1.

of which are presented in Section 4. The selection of a consistent and complementary set of models to generate, is based on a conceptual analysis of conceptual model types related to the different perspectives of a software system in development.

Our second research objective aims at developing a technique to generate the required models from related user stories as documented in a set of BDD scenarios. This development required two steps. First, we designed a framework and associated data structure for uniformly representing the information in the BDD scenarios. As theoretical foundation for the framework we used concepts from agent-oriented modelling. Our agent-based framework and structured representation of BDD scenarios are presented in Section 5. Second, we designed mappings from this structured representation to the concepts of stylized versions of UML diagrams that correspond to the selected conceptual models. These mappings were operationalized by defining mapping rules, which are presented along with the concept mappings in Section 6.

To demonstrate the model generation technique, a Natural Language Processing (NLP)-based tool was developed. This tool implements the creation of a structured representation of a set of related user stories based on the information in their BDD scenarios (and interpreted using the agent-based framework) and the algorithms for generating stylized UML diagrams from this structured representation. While demonstrating this tool in Section 7, some examples are elaborated that show how the generated models can be useful in Agile software development. As identifying and showing all use cases of the models requires a study on its own, the demonstration focuses on the use of the models for validating user stories and assuring requirements quality, leaving the exploration and demonstration of other use cases for future research.

To evaluate whether the models generated by our tool are also perceived as useful (and for which purposes) by practitioners, we conducted semi-structured interviews with eleven experts in Agile software development. This empirical evaluation is presented in Section 8. While the tool demonstration (Section 7) shows that our approach works given certain assumptions about the completeness and well-formedness of the BDD scenarios that are inputted to the tool, the empirical evaluation (Section 8) questions whether a consistent and complementary set of models can be generated that is perceived as useful for supporting Agile software development.

Regarding communication of the research results, the current paper is intended as a scholarly communication of our 'conceptual' solution that addresses the identified research gaps, while we aim at publishing in the future a more practitioner-oriented paper, focused on the actual use of the tool, as well as an academic paper that dives in the NLP-based techniques of our tool.

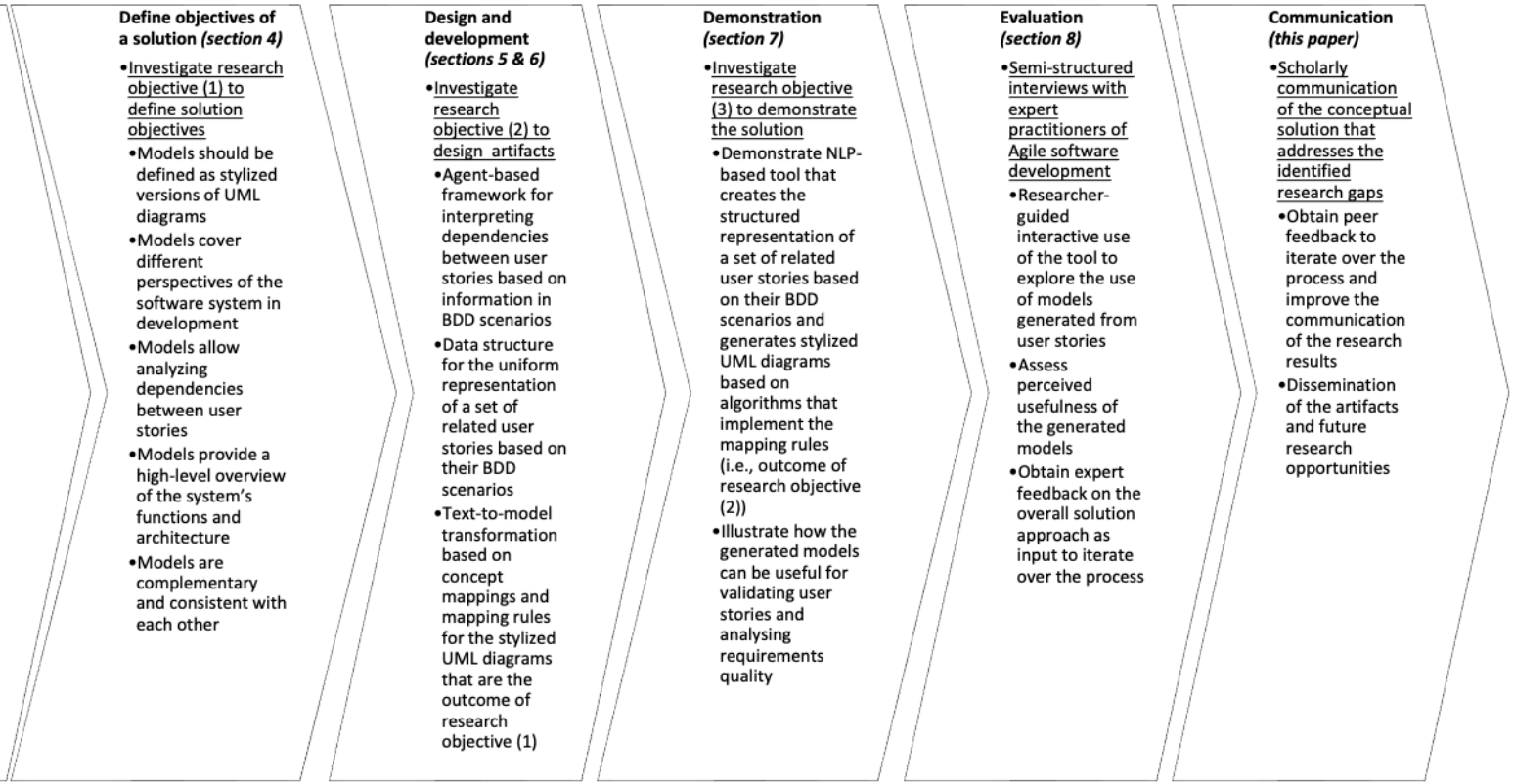

Figure . Design Science Research process (based on [33])

4 Selection of Conceptual Models

The solution objectives for our design artefacts are defined by the selection of conceptual models we wish to generate from user stories. From our problem investigation and the identified research gaps, we learned that to be potentially useful for supporting requirements analysis and system design activities in Agile software development practice, this selection needs to satisfy the following requirements regarding the consistency and complementarity of the generated conceptual models:

- The generated models should cover different perspectives of the software system in development – basically this means covering at least a structural perspective (i.e., what data to be stored by the system?) and a behavioral perspective (i.e., how to process the data to produce useful outcomes?);
- To be useful for requirements analysis, the selection of models should include models that allow analyzing dependencies between related user stories;
- To be useful for system design, the selection of models should include models that provide a high-level overview of the software system's functions and architecture;
- To facilitate their use and ensure consistency and complementarity, the models should be represented using a standard software engineering modelling language that supports different software perspectives with related types of models.

Guided by these requirements, our choice of modelling language is UML as this language supports the uniform modelling of different perspectives related to the structure and behavior of a software system that are relevant for systems analysis and design. Furthermore, UML is the standard language used by software engineers, which increases the chances that Agile software development practitioners are knowledgeable about the language. We therefore decided to select out of existing UML diagrams the types of conceptual model to generate from user stories, considering not only the requirements but also the information that is available in a set of BDD scenarios that document related user stories.

After a conceptual analysis of the different UML diagrams, we selected four different types of models: use case models (for what purposes systems are used by users), process models (what actions are performed by/for which users and in what order), domain models (what objects the system needs to store data about, how these objects are related, and which actions change their data), and state machines (how the state of objects, as represented by their data, changes through actions). These model types correspond respectively to the Use Case Diagram, Activity Diagram, Class Diagram, and State Machine Diagram included in UML.

The domain model (UML Class Diagram) is a structural diagram that shows the concepts of the domain and that serves as a base model for the software system's architecture. State machines (UML State Machine Diagram) offer a dynamic perspective of the objects modelled in the domain model showing how these objects change state through the functioning of the system. For every type of object modelled in the domain model (i.e., for every class in the UML Class Diagram), a state machine can be developed. Both types of models can further be used in the model-driven software development process [40].

While the domain model and state machines also offer opportunities for requirements analysis (i.e., acquiring and sharing an understanding of the structure and dynamics of a domain to be supported by a software system), two other types of conceptual model that are included in the selection are behavioral diagrams that support analyzing the relationships and dependencies between user stories. The use case model (UML Use Case Diagram) groups the functions required of the system as described in the user stories by user role, whereas the process model (UML Activity Diagram) can be used to analyze how the performance of functions depends on

the performance of other functions. To generate the models that show the dynamics of the system (i.e., process model and state machine), the information captured by BDD scenarios is needed.

A short description of the selected types of conceptual models is found in Table 2. These descriptions clarify the purpose and information content of the models and their relationships based on this purpose and content.

Conceptual model type	Description
Use Case Model	The UML Use Case Diagram is a conceptual model that shows how system users intend to use a system [41]. Each type of system use aimed at achieving some purpose is a *use case* and each type of direct or indirect system user is an *actor*. The model not only relates actors to use cases (by means of *associations*) but also use cases to other use cases to express relationships amongst those use cases (e.g., a use case may be included in another use case or extend another use case). Together, the use cases within the *system's boundary* describe what the system does and for which actors this is done [42].
Domain Model	The UML Class Diagram is used to show the concepts or different types of *object* (i.e., classes) that exist in a domain, their properties including *attributes* (i.e., class variables), *relationships* (i.e., associations between classes), and the *actions* (i.e., class methods) that can be performed on the objects [43]. Systems supporting a domain are used to store property values (i.e., data storage) and invoke actions (related to use cases as represented in the UML Use Case Diagram) that change these values (i.e., data processing).
Process Model	The UML Activity Diagram focuses on the *activities* performed by actors (modelled as *swimlanes*) to achieve some purpose in a domain of interest [44]. Activities represent actions on objects that are represented in the UML Class Diagram. The UML Activity Diagram is specifically used to model the *events* that trigger, interrupt or end these activities, and the *sequence flows* of the activities [45]. Non-sequential ordering of activities (e.g., choice, repetition, parallelism) is modelled through *gateways* (i.e., decision, fork, merge, and join nodes that implement XOR and AND logic operators for splitting or joining sequence flows).
State Machine	While the Process Model focuses on the sequencing of activities to achieve a purpose, an UML State Machine Diagram focuses on a single type of object (i.e., a class in the UML Class Diagram) and models how objects of this type change *state* through the *actions* modelled in the UML Activity Diagram [42]. The state of an object represents the values of its properties (i.e., the data that describes the object). Actions that change this state are represented as *transitions* [46].

Table 2: Selection of a consistent and complementary set of conceptual models to generate from a set of user stories

5 Theoretical Framework

Our first design artifact is a framework that helps interpreting the information captured in a set of related user stories that are written as BDD scenarios. This framework provides the theoretical basis for mapping the concepts of user stories to the concepts of the conceptual models selected in Section 4.

Models or ontologies that describe the concepts of user stories have been developed before. The model of Wautelet et al. [30] maps the three elements of the Connextra template on roles, activities or capabilities, and goals respectively. The model of Robeer et al. [47] distinguishes among the role, means, and ends parts of a user story as found in the Connextra template. A unified model of user stories, as per the Connextra template, and scenarios, as per the Gherkin format, is presented by Snoeck and Wautelet [40], based on Tsilionis et al. [48] and Wautelet et al. [49]. This model describes the concepts of BDD scenarios where a set of scenarios is associated to a user story, which differs from our definition of a BDD scenario as being composed of one user story (described using the Connextra template) and one scenario (described in the Gherkin format). All these proposed models describe the syntax of the templates used to document user stories (and scenarios in the case of the unified model). The semantic concepts found in these models (e.g., role, activity, goal, context, user behavior, outcome) are, however, not used to describe relationships and dependencies between user stories. Therefore, we developed a new model for interpreting the information about expected system functionalities captured in BDD scenarios, including dependencies and relationships between user stories.

As theoretical framework for this model we use concepts related to agent-oriented modelling [50]. An extensive literature exists about agents that describes various properties of agents. An agent acts in an environment (e.g., an organization) and can perform actions that will affect itself and its environment [51]. An agent can respond to stimuli and make choices among possible actions to achieve goals [52]. Wooldridge [53] describes agents as being autonomous, exhibiting goal-directed (proactive) behavior, responding to changes in the environment (reactive), and being able to interact. By understanding the role element of a user story as an agent, elements of BDD scenarios can be interpreted as agent-related concepts which will elucidate relationships and dependencies between user stories as we will explain further on in this section.

Based on this description of agents and related concepts, we propose in Table 3 a set of propositions. These propositions define the agent-based framework that we use to model a set of BDD scenarios that document related user stories. Figure 2 visualizes these propositions using an Extended Entity-Relationship (EER) Diagram. The constraint that the postcondition of an action should include the object on which the action is performed, and the intended state change of that action (part of proposition 6) cannot be shown due to limitations of the semantic expressiveness of EER Diagrams. Also, proposition 5 is not shown as it requires a behavioral diagram to represent state changes.

No.	Propositions
1	Agents operate in an organizational environment where they perform actions on objects to achieve goals.
2	An action is always performed by one agent. Agents can perform many actions.
3	An action is always performed on one object. Many actions can be performed on an object.
4	An object has one state at any given time.
5	An action performed on an object may cause a state transition for that object.

6	A postcondition of an action is comprised of one or more objects and their corresponding states after the action is performed. The postcondition should at least include the object on which the action is performed and the intended state change of that action.
7	A precondition of an action is comprised of zero or more objects and their corresponding states before the action can be performed.
8	An object in a state can be part of many preconditions and many postconditions.

Table 3. Agent-related propositions for BDD scenarios

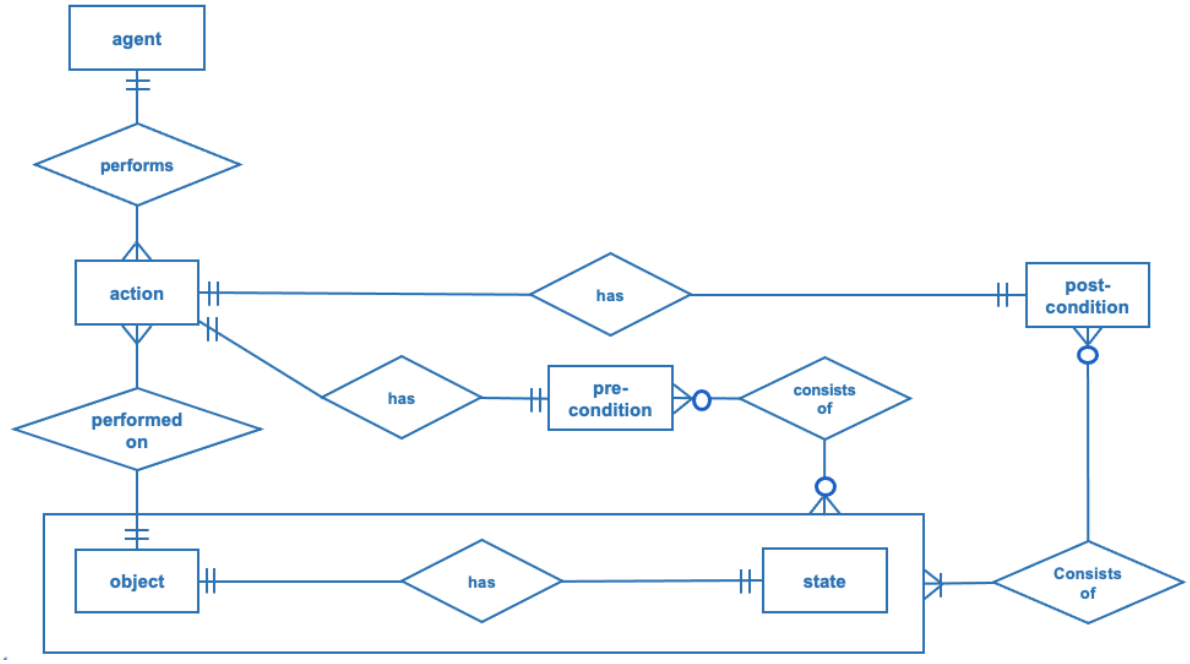

Figure 2. Agent-Based Framework for BDD Scenarios

We now map the elements of the BDD scenario template defined in Section 2 (Table 1) to the concepts of the agent-based framework as follows:

- The *role* referring to the (type of) user is mapped to the *agent* concept. In a BDD scenario, a (type of) user wishes to achieve certain goals using the system, so is an agent.
- The *means* is mapped to an *action* performed on an *object*.
- The *ends* is not mapped to a concept of the agent-based framework, as the concept of goal is not defined. Nevertheless, whatever the goal might be, the effect of the means is described in the postcondition.
- The *context (and/or some more context)* is mapped to a *precondition*, which consists of zero, one or many *objects* in certain *states*. In the exceptional case that no context is specified we make use of an empty precondition (i.e., one that consists of no objects in a certain state).
- *Some event occurs (and/or some other event occurs)* is not mapped to a concept of the agent-based framework, as the concept of triggering event is not defined.
- The *outcome (and some other outcome)* is mapped to a *postcondition*, which consists of one or many *objects* in certain *states*.

Table 4 illustrates this mapping for the example BDD scenario that was introduced in Section 2 (Table 1).

BDD Scenario	Concepts of the Agent-Based Framework	Instances of Concepts of the Agent-Based Framework
As a customer,	Agent (that performs the action)	customer (agent)
I want to cancel a service request	Action performed on Object	cancel (action) service request (object)
so that the team can focus on other active requests.	/	/
Given a service request is submitted or fixed,	Precondition (of action) consisting of zero or more Object has State pairs	service request (object) submitted (state), service request (object) fixed (state)
when the customer decides to cancel the service request,	/	/
then the service request should be cancelled.	Postcondition (of action) consisting of one or more Object has State pairs	service request (object) cancelled (state)

Table 4. Example mapping of BDD Scenario elements on the concepts of the Agent-Based Framework

What sets this model apart from the reviewed models is that it shows that user stories involve performing actions on objects. These actions change the state of the objects. Some actions are only allowed when objects are in a particular state. If we now consider a set of related user stories, then the actions of those user stories may have agents, objects, (parts of) preconditions and (parts of) postconditions in common. In such a set of user stories that have a common objective of defining proposed functionality for the system in focus, agents collaborate to achieve common goals. Through the actions of agents, object states are transitioned towards intended states that achieve those common goals. Postconditions of actions will be (part of) preconditions of other actions, thus indicating dependencies between the user stories that are considered. Hence, modelling preconditions and postconditions of user stories as specified in BDD scenarios in terms of object-state pairs allows identifying dependencies between user stories. User stories can also be related in other ways, for instance, user stories are related if they concern the same agent or if they involve actions on a same object.

To identify dependencies and relationships between user stories, it is useful to represent entire sets of related user stories in terms of the agent-based framework. To this end, we propose based on the agent-based framework a structured representation of a set of BDD scenarios that document related user stories. This structured representation will serve as a uniform data format for inputting the information contained in the BDD scenarios to the algorithms that generate conceptual models (see Section 6).

To demonstrate the proposed structured representation, we work with an example set of four BDD scenarios (Table 5). For the sake of genericity, we formulate these BDD scenarios in abstract, domain/application-independent terms. If the user stories documented in these BDD scenarios are related, for instance, are part of the same Scrum feature or epic containing interdependent functions, then many of these terms will refer to the same instance of a concept in the agent-based framework (e.g., <object_1> = <object_1A>). We will state an example of these equivalences in Section 6 where the same set of BDD scenarios is used to illustrate the mapping onto conceptual models.

We also include in Table 5 an instance of each of the four basic types of BDD scenarios, based on a possible disjunction or conjunction of object-state pairs in the precondition and a possible conjunction of object-state pairs in the postcondition, conforming to our assumptions regarding the use of the BDD scenario template as defined in Section 2 (Table 1). All possible BDD scenarios can be created by combining instances of these basic types. Also, but not included in the examples, conjunctions and disjunctions of more than two object-state pairs are possible. We do not consider more complex situations where a precondition involves a combination of disjunctions and conjunctions of object-state pairs.

BDD scenario	Example
Simple scenario	**BDD scenario 1:** As an <agent_1>, I want to perform <action_1> on <object_1> so that <goal_1> is achieved. Given <object_1A> is in <state_1A>, when <action_1> is triggered, then <object_1> is in <state_1B>.
With conjunction (AND) in the postcondition	**BDD scenario 2:** As an <agent_2>, I want to perform <action_2> on <object_2> so that <goal_2> is achieved. Given <object_2A> is in <state_2A>, when <action_2> is triggered, then <object_2> is in <state_2B> and <object_2C> is in <state_2C>.
With conjunction (AND) in the precondition	**BDD scenario 3:** As an <agent_3>, I want to perform <action_3> on <object_3> so that <goal_3> is achieved. Given <object_3A> is in <state_3A> and <object_3B> is in <state_3B>, when <action_3> is triggered, then <object_3> is in <state_3C>.
With disjunction (OR) in the precondition	**BDD scenario 4:** As an <agent_4>, I want to perform <action_4> on <object_4> so that <goal_4> is achieved. Given <object_4A> is in <state_4A> or <object_4B> is in <state_4B>, when <action_4> is triggered, then <object_4> is in <state_4C>.

Table 5. Basic types of **BDD scenarios** and abstracted examples

To create the structured representation of a set of BDD scenarios, we define a table (Table 6Table) with columns agent, action, object, precondition object, precondition state, postcondition object and postcondition state. Each BDD scenario in the set is represented by one or more rows in this table, based on the mapping of the elements of the BDD scenarios on the concepts of the agent-based framework. An instance of a simple BDD scenario will occupy exactly one row in the table. However, when there are multiple object-state pairs in the precondition or postcondition, additional row(s) is(are) used, keeping all other column values unchanged. The logical operators AND and XOR are added to all precondition or postcondition objects that are part of multiple object-state pairs in preconditions and postconditions respectively.

BDD scenario	Agent	Action	Object	Pre-condition Object	Pre-condition State	Post-condition Object	Post-condition State
1	agent_1	action_1	object_1	object_1A	state_1A	object_1	state_1B
2	agent_2	action_2	object_2	object_2A	state_2A	object_2 (AND)	state_2B
2	agent_2	action_2	object_2	object_2A	state_2A	object_2C (AND)	state_2C
3	agent_3	action_3	object_3	object_3A (AND)	state_3A	object_3	state_3C
3	agent_3	action_3	object_3	object_3B (AND)	state_3B	object_3	state_3C
4	agent_4	action_4	object_4	object_4A (XOR)	state_4A	object_4	state_4C
4	agent_4	action_4	object_4	object_4B (XOR)	state_4B	object_4	state_4C

Table 6. Structured representation of a set of abstract BDD scenarios

6 Generating Conceptual Models from BDD Scenarios

Our second design artefact is a set of concept mappings, operationalized by mapping rules, from the structured representation of a set of BDD scenarios (Section 5) to the conceptual models that we selected (Section 4). We present these models here as stylized versions of the selected UML diagrams (i.e., Use Case Diagram, Activity Diagram, Class Diagram, and State Machine Diagram) as not all model constructs for these diagram types are used in the mapping. The stylized diagrams are still recognizable as their corresponding standard UML diagrams, however, because of the constraint that all information to generate the models should be present in the BDD scenarios, allowing for the automation of the mapping rules, the abstract syntax of the standard UML diagrams is simplified.

Before presenting the concept mappings and mapping rules, we first extend the abstracted examples of Table 6. As we generate conceptual models for a set of BDD scenarios that document related user stories, these user stories are aggregated by the same Scrum epic or theme (or equivalent Agile development construct) and share at least some of the instances of the concepts of the agent-based framework. We therefore state in Table 7Table Table several equivalences of concept instances for the sake of demonstrating the generation of conceptual models.

Assumed equivalences of agent-based framework concept instances
agent_1 = agent_3
object_1 = object_1A = object_2C =object_3 = object_3A = object_4 = object_4A = object_4B
object_2 = object_2A = object_3B
state_1B = state_3A = state_4A
state_2B = state_3B
state_2C = state_4B

Table 7. Equivalent concept instances for the abstracted examples of Table 6

Applying the equivalences of concept instances of Table 7Table to the structured representation of Table 6 results in Table 8, which is the structured representation that we will use to illustrate in the next four sub-sections the concept mappings and mapping rules for the four different types of stylized UML diagram that we selected.

BDD scenario	**Agent (1)**	**Action (2)**	**Object (3)**	**Precondition**		**Postcondition**	
				Object (4)	**State (5)**	**Object (6)**	**State (7)**
1	agent_1	action_1	object_1	object_1	state_1A	object_1	state_1B
2	agent_2	action_2	object_2	object_2	state_2A	object_2 (AND)	state_2B
2	agent_2	action_2	object_2	object_2	state_2A	object_1 (AND)	state_2C
3	agent_1	action_3	object_1	object_1 (AND)	state_1B	object_1	state_3C
3	agent_1	action_3	object_1	object_2 (AND)	state_2B	object_1	state_3C
4	agent_4	action_4	object_1	object_1 (XOR)	state_1B	object_1	state_4C
4	agent_4	action_4	object_1	object_1 (XOR)	state_2C	object_1	state_4C

Table 8. Structured representation of a set of abstract BDD scenarios (modified from Table 6 for illustrating the model generation mapping rules)

7.1 Use Case Model

In the proposed agent-based framework, agents perform actions on objects to achieve goals (i.e., proposition 1 of the Agent-based Framework (Table 1)). A use case model expresses what actors expect as functionality, represented as a use case, from a system. Hence, we map agents to actors and actions performed on objects to use cases (Table 9).

The system boundary construct is not used in the stylized use case diagrams as we generate models from a set of user stories that are related, hence they are assumed to capture

requirements related to a same system. We further assume that such related set of user stories is described at a level of granularity that restricts the scope of each user story to a single type of user performing a single action on one specific object (see our assumptions for the use of the BDD scenario template in sub-section 2.2). It is therefore unlikely that these fine-grained user stories may contain the information needed to derive includes and extends relationships, which would require defining user stories at different granularity levels (as for instance proposed by Wautelet et al. [30]). We believe that a use case diagram with only actors, use cases, and their associations is essentially still a use case model that provides a graphical overview of user roles and types of system use, so captures at a high-level of abstraction the system's required functionality as grouped by user roles.

Structured Representation of a Set of BDD Scenarios	**Use Case Diagram**
Agent	Actor
Action - Object	Use Case

Table 9. Concept mapping for the structured representation of a set of BDD Scenarios and Use Case Diagram

To perform the mapping of concepts such that a use case diagram can be generated, the following rules are proposed. The third rule is needed to link actors to use cases.

- **Mapping rule 1.1:** For each *unique* agent in the structured representation of a set of BDD scenarios, add an actor to the use case diagram.
- **Mapping rule 1.2:** For each *unique* action – object combination (i.e., the action is performed on the object) in the structured representation of a set of BDD scenarios, add a use case to the use case diagram.
- **Mapping rule 1.3:** For each agent that performs an action on an object in the structured representation of a set of BDD scenarios, draw an association between the actor that represents the agent and the use case that represents the action-object combination.

The use case diagram shown in Figure 3 was obtained by applying these mapping rules to the example structured representation of Table 8.[2] There are three unique agents in the structured representation (i.e., agent_1, agent_2 and agent_4), hence these three agents are modelled as actors in the use case diagram. There are four unique combinations of actions performed on objects in the structured representation (i.e., one per BDD scenario), so these are shown as use cases. Agent_1 is associated to two of those use cases, while the other agents are associated each to a single use case.

[2] The stylized UML diagrams shown as illustration in this section are similar to the diagrams that are automatically generated by the tool that executes the algorithms for implementing the mapping rules. The concrete syntax used does not fully conform to UML. The application of the tool to a real set of BDD scenarios is demonstrated in Section 7.

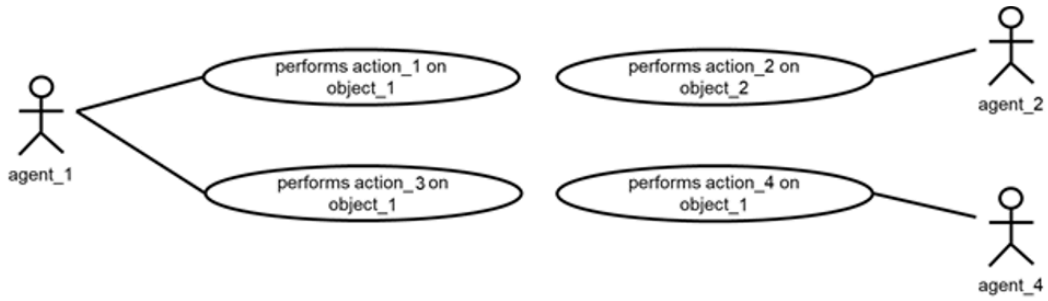

Figure 3. Use case diagram created from the example structured representation of Table 8.

7.2 Domain Model

In the proposed agent-based framework, objects have states (i.e., proposition 4 of the Agent-based Framework (Table 3)), so correspond to entities in the domain model. We can derive from the definition of agents that they also have states, even if we decided that the modelling of agent states is currently out of scope of the agent-based framework. Hence, in a domain model, agents can also be modelled as entities. Actions link agents to objects (i.e., propositions 2 and 3 of the Agent-based Framework (Table 3)), so we can model them as relationships between agent entities and object entities in the domain model. If the domain model is represented as a class diagram, then entities and relationships are modelled as classes and associations respectively (Table10).

Since we do not assume that BDD scenarios are specified at a level of detail where individual data attributes are identifiable (instead we assume a more abstract description of object states as explained in sub-section 2.2), the stylized class diagrams will not include class variables. Information on actions is available, however, we decided not to model actions as class methods in our stylized class diagrams as actions are already captured by the other types of conceptual models. Discovering multiplicities in user stories and associated BDD scenarios will be challenging (and we are not sure this is possible at all). Further, we decided not to include semantic relationships between classes such as specialization, aggregation, and composition as this requires advanced NLP techniques for the interpretation of the semantics of user roles and objects, which is outside our current scope of research. Hence, the stylized class diagrams only include classes and associations, where classes model the domain concepts and associations the relationships between these domain concepts. Domain concepts such as user roles and objects, and relationships such as a user role using or expecting the system to perform an action on an object, can thus be shown in the graphical overview of the domain provided by the stylized class diagram.

Structured Representation of a Set of BDD Scenarios	**Class Diagram**
Agent	Class
Object	Class
Action	Association

Table 10. Concept mapping for the structured representation of a set of BDD Scenarios and Class Diagram

Based on this concept mapping, we propose the following rules for mapping a set of BDD scenarios to a class diagram.

- **Mapping rule 2.1:** For each *unique* agent in the structured representation of a set of BDD scenarios, add a class to the class diagram.
- **Mapping rule 2.2:** For each *unique* object in the structured representation of a set of BDD scenarios, add a class to the class diagram.
- **Mapping rule 2.3:** For each agent that performs an action on an object the structured representation of a set of BDD scenarios, draw an association between the class that represents the agent and the class that represents the object. Label the association with the action performed by the agent on the object.

Applying these mapping rules to the example structured representation (Table 8) results in Figure 4. We can see that agent_1 is linked to object_1 via two associations, representing action_1 and action_3. Deviating from standard UML, only one association line is drawn, however, this association line has two labels (i.e., action_1 and action_3) signifying the presences of two associations. Further, Agent_4 is also related to object_1, now via the association that represents action_4. Finally, agent_2 is related via action_2 to object_2.

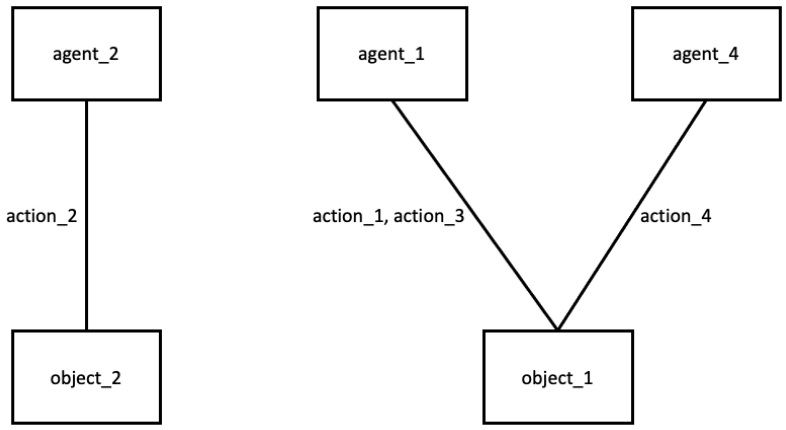

Figure 4. Class diagram created from the example structured representation of Table 8.

7.3 State Machine

The concept mapping for state machine diagrams is shown in Table 11. For each object in the structured representation of a set of BDD scenarios on which an action is performed, a state machine is modelled. The states in these state machines are the precondition and postcondition states of these objects in the BDD scenarios. Actions performed on objects result in state transitions (i.e., proposition 5 of the Agent-based Framework (Table 3)). We did not investigate the use of other UML concepts for modelling state machines (e.g., entry/do/exit activities). Even without these concepts, the stylized state machine diagrams show how related user stories (as documented in a set of BDD scenarios) move objects through different states such that agents achieve their goals.

Structured Representation of a Set of BDD Scenarios	**State Machine Diagram for Object$_i$ ($i \in \{1, .., n\}$, with n = number of unique objects on which actions are performed in the structured representation of a set of BDD scenarios)**
Precondition State of Precondition Object$_i$	State
Postcondition State of Postcondition Object$_i$	State
Action on Object$_i$	Transition

Table 11. Concept Mapping for the structured representation of a set of BDD Scenarios and State Machine Diagrams

The following three mapping rules are directly derived from this concept mapping.

- **Mapping rule 3.1:** For each *unique* object on which an action is performed in the structured representation of a set of BDD scenarios, create a state machine diagram.
- **Mapping rule 3.2:** For each *unique* precondition or postcondition state of a specific object in the structured representation of a set of BDD scenarios, add a state to the corresponding state machine diagram.
- **Mapping rule 3.3:** If a state of an object is in the precondition of a BDD scenario and a different state of the same object is in the postcondition of the same BDD scenario, draw a transition from the state that represents the precondition state to the state that represents the postcondition state in the state machine diagram for that object. The label of the transition is the action performed on the object.

It is common to indicate initial states and final states on UML state machine diagrams (although such states can also be identified as states with no incoming, respectively no outgoing transitions). We therefore introduce mapping rules 3.4 and 3.5. Further, to apply the concept mapping correctly and completely, an additional mapping rule is needed. Whereas an action on an object as described in a BDD scenario always changes the state of that object (i.e., proposition 6 of the Agent-based Framework (Table 3)), the state of an object can also change because of an action performed on another object, as described in another BDD scenario. To model this in the state machine diagram of an object, we introduce 'unknown' states in mapping rule 3.6. Note that mapping rules 3.4, 3.5 and 3.6 are only applied for objects for which state machine diagrams are created (i.e., mapping rule 3.1). In other words, we do not create state machine diagrams for objects whose state is changed just as a side-effect of the set of BDD scenarios considered.

- **Mapping rule 3.4:** If a state of an object is in the precondition of a BDD scenario and this state is not in the postcondition of any other BDD scenario, then indicate in the state machine diagram for the object that this is an initial state.
- **Mapping rule 3.5:** If a state of an object is in the postcondition of a BDD scenario, and this state is not in the precondition of any other a BDD scenario or it is in the precondition of another BDD scenario but not in the postcondition of that other a BDD scenario, then indicate in the state machine diagram for the object that this is a final state.
- **Mapping rule 3.6:** If a state of an object is in the postcondition of a BDD scenario and this object is not in the precondition of the same BDD scenario, then create a 'unknown' state in the state machine diagram of the object and draw a transition from this 'unknown' state to the state representing the postcondition state in the state machine diagram for the object. The label of the transition is the action of the user story documented in the BDD scenario.

Applying these mapping rules to the example structured representation (Table 8) results in Figure 5. Mapping rule 3.1 creates state machine diagrams for object_1 (top of Figure 5) and object_2 (bottom of Figure 5). Mapping rule 3.2 adds state_1A, state_1B, state_2C, state_3C and state_4C as states to the state machine diagram for object_1. The same mapping rule adds states state_2A and state_2B to the state machine diagram for object_2. Mapping rule 3.3 adds transitions from state_1A to state_1B (label: 'action_1'), from state_1B to state_3C (label: 'action_3'), from state_1B to state_4C (label: 'action_4'), and from state_2C to state_4C (label: 'action_4') in the state machine diagram for object_1, and a transition from state_2A to state_2B (label: 'action_2') in the state machine diagram for object_2. Mapping rule 3.4 identifies state_1A and state_2A as initial states. Mapping rule 3.5 identifies as final states, state_3C and state_4C in the state machine diagram for object_1 and state_2B in the state machine diagram for object_2. Finally, mapping rule 3.6 adds an unknown state to the state machine diagram for object_1 and a transition from that unknown state to state_2C with label 'action_2'.

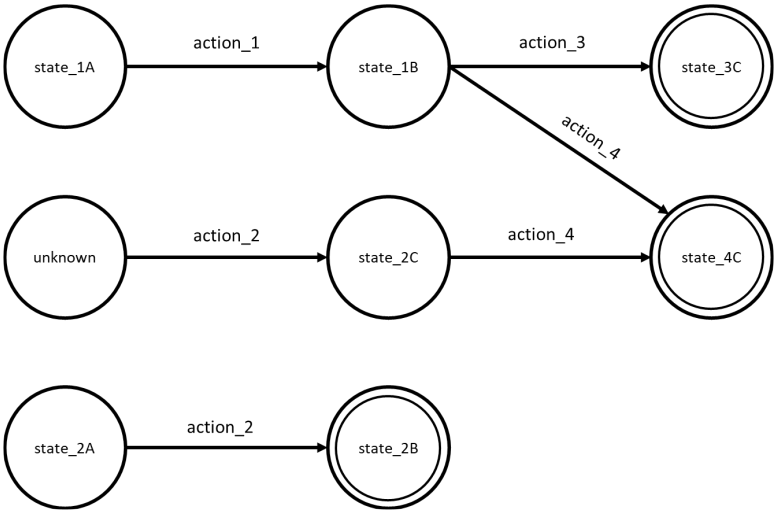

Figure 5. State machine diagrams created from the example structured representation of Table 8.

7.4 Process Model

We use an activity diagram to show dependencies between the user stories that are documented in a set of BDD scenarios. Within a set BDD scenarios, a user story *x* depends on a user story *y*, if the action of *y* needs to be performed as a condition for performing the action of *x*, which is information we can derive from the BDD scenarios. For modelling these dependencies, the activity diagram uses control flows. As we will show, such conditions might be combined in a conjunction or disjunction as allowed by the BDD scenario template (Table 1) and our assumptions regarding its use as discussed in Section 2. These conditions might also be absent. There can also be user stories that are not depended on by other user stories. We will show that all this information is captured by the structured representation of a set of BDD scenarios and can be represented in the activity diagram using start nodes and end nodes, and using the split and join, and the AND and XOR semantics, of the UML merge nodes (XOR-join), join nodes (AND-join), and fork nodes (AND-split). As the main purpose we assign to the process model is to show dependencies between user stories, we did not investigate whether more advanced UML concepts for modeling data flow (e.g., object node, object flow) and temporal behavior (e.g., send signal action, accept change event action, accept time event action) could be used in the mapping.

To create a stylized activity diagram from a structured representation of a set of BDD scenarios, we use the concept mapping presented in Table 12. According to proposition 1 in the Agent-based Framework (Table 3), agents perform actions (using the system for which the user stories are formulated), hence agents map to swimlanes in activity diagrams. Actions are performed on objects (i.e., proposition 3 of the Agent-based Framework (Table 3)) and are always performed by one agent (i.e., proposition 2 of the Agent-based Framework (Table 3)). Accordingly, the action-object pair in a BDD scenario is mapped onto the UML concept of activity, where an activity is always positioned within one swimlane (that represents the agent performing the action on the object).

A match between the state of an object in the postcondition of one BDD scenario and the same object in the same state in the precondition of another BDD scenario (i.e., proposition 8 of the Agent-based Framework (Table 1)), is mapped to a control flow, as this means that after the action of the first user story is performed, the action of the second user story can possibly be performed. If no such match can be found for an object in a state in a precondition of a BDD scenario, then a start node is identified. Similarly, if no such match can be found for an object in a state in a postcondition of a BDD scenario, then an end node is identified. How these events are connected to the other elements of the activity diagram, is explained in the mapping rules (confer infra).

Finally, if there is more than one object-state pair in the precondition or postcondition of a BDD scenario (i.e., propositions 6 and 7 of the Agent-based Framework (Table 3)), then a non-sequential control flow needs to be modelled preceding, respectively succeeding the activity that represents the action of the user story. The precise nature of the non-sequential control flow (i.e., AND or XOR, join or split semantics) is determined by the mapping rules (confer infra).

Structured Representation of a Set of BDD Scenarios	**Activity Diagram**
Agent	Swimlane
Action - Object	Activity
Postcondition State of Postcondition Object in a BDD scenario *is equal to* Precondition State of Precondition Object in *some other* BDD scenario	Control Flow
Precondition State of Precondition Object in a BDD scenario *is not equal to* Postcondition State of Postcondition Object in *any other* BDD scenario	Start Node
Postcondition State of Postcondition Object in a BDD scenario *is not equal to* Precondition State of Precondition Object in *any other* a BDD scenario	End Node
Multiple Object – State pairs in the Precondition of a BDD scenario	XOR-join (Merge Node) AND-join (Join Node)
Multiple Object – State pairs in the Postcondition of a BDD scenario	AND-split (Fork Node)

Table 12. Concept mapping for the structured representation of a set of BDD Scenarios and Activity Diagram

We present and illustrate the mapping rules in four stages. In the first stage, activities and swimlanes are identified. In the second stage, activities are connected by control flows. In the third stage, start and end nodes are identified. In the fourth stage, non-sequential control flow is identified and inserted in the model using the appropriate UML constructs.

The rules for the first stage of the mapping are:

- **Mapping rule 4.1:** For each *unique* agent in the structured representation of a set of BDD scenarios, add a swimlane to the activity diagram.
- **Mapping rule 4.2:** For each *unique* action – object pair (i.e., the action is performed on the object) in the structured representation of a set of BDD scenarios, add an activity to the activity diagram in the swimlane for the agent who performs the action on the object.

Appling these two mapping rules to the example structured representation (Table 8)Table results in Figure 6, which shows that agent_1 performs action_1 and action_3, agent_2 performs action_2, and agent_4 performs action_4. Note that the labels of the actions should also mention the object on which the action is performed (e.g., action_1 object_1) but we omit this here in order not to overload the diagrams which are just used for the sake of illustration.

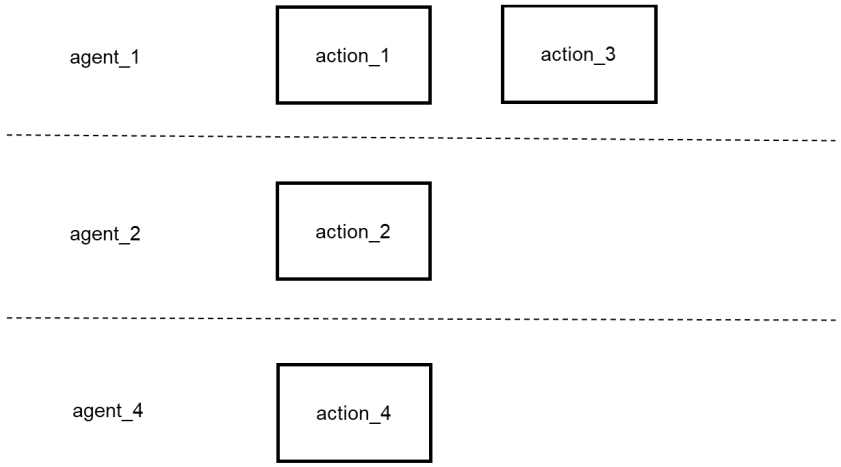

Figure 6. Activity diagram (Stage 1) created from the structured representation of Table 8

The mapping rule for the second stage is:

- **Mapping rule 4.3:** If the state of an object in the postcondition of a BDD scenario is equal to the state of that object in the precondition of *another* BDD scenario, then add a control flow from the activity corresponding to the action – object pair of the first user story to the activity corresponding to the action – object pair of the second user story.

We now apply this mapping rule to the example structured representation (Table 8Table) to add control flows to the Stage 1 activity diagram. The resulting State 2 activity diagram is shown in Figure 7. In total, four control flows were identified as matches were found between rows 1 (BDD scenario 1) and 4 (BDD scenario 3), rows 1 (BDD scenario 1) and 6 (BDD scenario 4), rows 2 (BDD scenario 2) and 5 (BDD scenario 3), and rows 3 (BDD scenario 2) and 7 (BDD scenario 4).

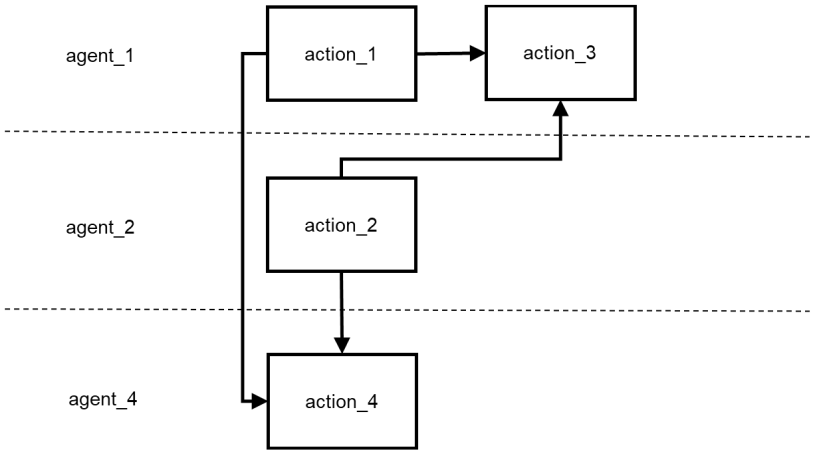

Figure 7. Activity diagram (Stage 2) created from the structured representation of Table 8

In the third stage of generating the activity diagram, start and end nodes are identified. This can be done using the following mapping rules:

- **Mapping rule 4.4:** If the state of an object in the precondition of a BDD scenario is not equal to the state of that object in the postcondition of *any other* BDD scenario, then add a start node to the activity diagram and connect it with a control flow to the activity representing the action described in the BDD scenario which precondition was considered.
- **Mapping rule 4.5:** If the state of an object in the postcondition of a BDD scenario is not equal to the state of that object in the precondition of *any other* BDD scenario, then add an end node to the activity diagram and connect the activity representing the action described in the BDD scenario which postcondition was considered with a control flow to this end node.

The result of applying these mapping rules to the example structured representation (Table 8Table) is the Stage 3 activity diagram that is shown in Figure 8. Two start nodes were identified because of the unmatched preconditions in rows 1 (BDD scenario 1) and 2 (BDD scenario 2). Two end nodes were identified because of the unmatched postconditions in rows 4 (BDD scenario 3) and 6 (BDD scenario 4). Note that the notation for start and end nodes in Figure 8 differs slightly from the standard UML notation.

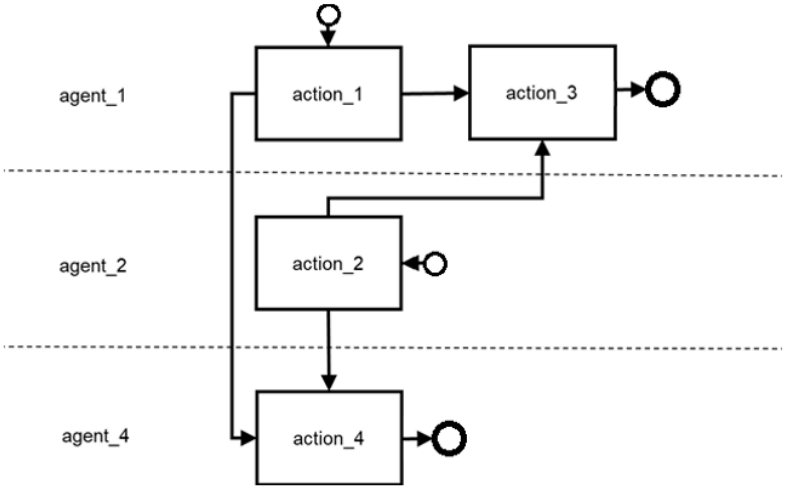

Figure 8. Activity diagram (Stage 3) created from the structured representation of Table 8

Finally, we present the mapping rules to introduce non-sequential control flow in the activity diagram. These rules consider the logical operators AND and XOR that were added in the structured representation of a set of BDD scenarios to all precondition or postcondition objects that are part of object-state pairs in preconditions, respectively postconditions with multiple object-state pairs.

- **Mapping rule 4.6:** If there is a disjunction of object-state pairs in the precondition of a BDD scenario, then add a merge node (i.e., XOR-join) *before* the activity representing the action-object pair of the user story. This merge node merges the incoming control flows of the activity.
- **Mapping rule 4.7:** If there is a conjunction of object-state pairs in the precondition of a BDD scenario, then add a join node (i.e., AND-join) *before* the activity representing the action-object pair of the user story. This join node merges the incoming control flows of the activity.
- **Mapping rule 4.8:** If there is a conjunction of object-state pairs in the postcondition of a BDD scenario, then add a fork node (i.e., AND-split) *after* the activity representing the action-object pair of the user story. This fork node splits the outgoing control flows of the activity.

Applying these mapping rules to the example structured representation (Table 8Table) results in the Stage 4 activity diagram that is shown in Figure 9, which is the final process model resulting from the mapping. Note that the symbols for merge, join, and fork nodes deviate from standard UML. The merge node is modelled using an empty diamond with two or more incoming control flows and one outgoing control flow, similar to the XOR-join gateway symbol of BPMN. The join node has as symbol a diamond with a plus sign and two or more incoming control flows and one outgoing control flow, similar to the AND-join gateway in BPMN. Finally, the fork node also has a plus sign inside diamond symbol but now with two or more outgoing control flows and a single incoming control flow, like the AND-split gateway of BPMN.

BDD scenario 4 states that action_4 is performed on object_1 if that object is in state_1B or state_2C. These states are the outcomes of respectively action_1 (BDD scenario 1) and action_2 (BDD scenario 2). Mapping rule 4.6 adds a merge node (i.e., XOR- join semantics of the non-sequential control flow) to the activity diagram that merges the control flows coming from these two actions into one control flow going into action_4. This merge node thus indicates that action_4 is preceded by either action_1 or action_2.

BDD scenario 3 specifies that action_3 is performed on object_1 if object_1 is in state_1B and object_2 is in state_2B, which are the outcomes of respectively action_1 (BDD scenario 1) and action_2 (BDD scenario 2). Mapping rule 4.7 adds a join node (i.e., AND- join semantics of the non-sequential control flow) to the activity diagram that merges the control flows coming from action_1 and action_2 into one control flow going into action_3. The activity diagram thus specifies that action_3 is preceded by both action_1 and action_2.

Finally, BDD scenario 2 states that the outcome of action_2 is object_2 in state_2B and object_1 in state_2C. Mapping rule 4.8 inserts a fork node (i.e., AND-split semantics of the non-sequential control flow) into the activity diagram that splits a control flow going out of action_2 into two separate control flows, each signifying the context of an object in a particular state, respectively object_2 in state_2B and object_1 in state_2C. The first control flow coming out of this fork node goes into the join node that was added by mapping rule 4.7 as object_2 in state_2B is part of the conjunction in the precondition of BDD scenario 3, hence action_3 must be preceded by action_2. The second control flow coming out of the fork node goes into the

merge node that was added by mapping rule 4.6 as object_1 in state_2C is part of the disjunction in the precondition of BDD scenario 4, hence action_2 results in a context in which action_4 can be performed but is not the only action that fulfills the precondition of action_4.

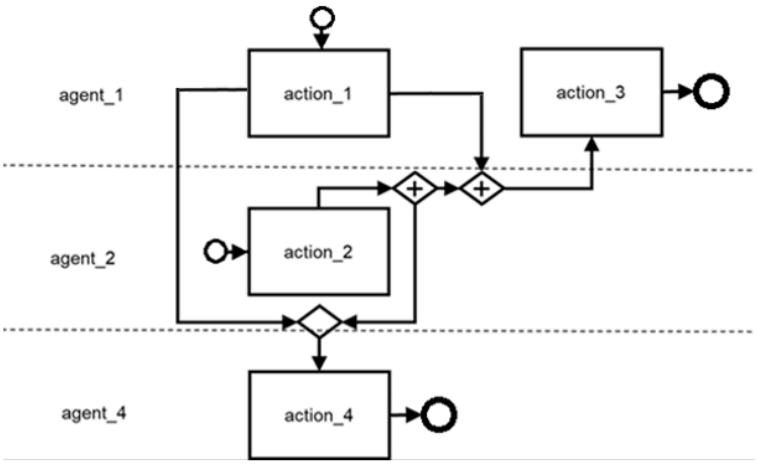

Figure 9. Final activity diagram (Stage 4) created from the structured representation (Table 8)

7 Demonstration and Proof of Concept

Our third design artefact are the algorithms that implement the mapping rules defined in Section 6. These algorithms were coded in a software tool that first employs NLP-techniques to create a structured representation of a set of BDD scenarios that document related user stories, as proposed in Section 5, and next executes the algorithms to generate the stylized UML diagrams corresponding to the conceptual models selected in Section 4. Using this tool, a use case diagram, class diagram, activity diagram, and state machine diagrams can be generated from a set of BDD scenarios if these are written using the BDD scenario template presented in Table 1 and the assumptions hold regarding the use of this template as discussed in sub-section 2.2.

The tool is conceived as a working prototype that can *automatically* generate conceptual models as output when BDD scenarios (for which our assumptions hold) are provided as input. The tool was developed in Python. Guidelines on how to use the tool (Setup.txt), sample user stories (Input.txt), and sample models (i.e., stylized UML diagrams) as outputs can be found online.[3]. The pseudo-code algorithms for implementing the mapping rules can also be found here and in Appendix A.

To demonstrate the tool, we inputted the following set of BDD scenarios:

1. *As a customer, I want to create a service request so that I can have my problem solved. Given that the customer is active, when they decide to submit a service request, then a service request should be submitted.*
2. *As a support assistant, I want to accept a service request so that the team can start working on the service request. Given a service request is submitted, when the team agrees working on the service request, then the service request should be open.*

[3] https://github.com/conceptualmodel/modelsfromuserstory

3. *As a support assistant, I want to resolve a service request so that the customer's problem is solved. Given a service request is open, when the team solves the problem described in the service request, then the service request should be fixed.*
4. *As a customer, I want to approve a service request so that it can be closed. Given a service request is fixed, when I approve the solution to my problem, then the service request should be closed.*
5. *As a customer, I want to reject a service request so that it can be reopened. Given a service request is fixed, when I reject the solution to my problem, then the service request should be open.*
6. *As a customer, I want to cancel a service request so that the team can focus on other active requests. Given a service request is submitted or fixed, when the customer decides to cancel the service request, then the service request should be canceled.*

These BDD scenarios document user stories that could for instance be a Scrum theme that describes required functionality of a software system supporting service request management (e.g., a system supporting the ITIL 4 service request management practice [35]). Note that BDD scenario 6 was already introduced as example user story and scenario in Section 2. The user roles in this theme are customer and support assistant. The overall goal is to develop a workflow driven application to handle customer service requests. The scope of the theme is deliberately kept small to enable traceability between the generated models and the BDD scenarios they are generated from.
The tool creates as an internal data structure the structured representation of the inputted BDD scenarios (Table 13). There are two rows in this data structure for BDD scenario 6 as its precondition has two object-state pairs that are combined in a disjunction. The context for performing the cancel action on the service request object is either the service request object in state submitted or the service request object in state fixed. All other BDD scenarios are simple scenarios as in the abstract example of Table 5.

BDD scen.	**Agent (1)**	**Action (2)**	**Object (3)**	**Precondition**		**Postcondition**	
				Object (4)	**State (5)**	**Object (6)**	**State (7)**
1	customer	create	service request	customer	active	service request	submitted
2	support assistant	accept	service request	service request	submitted	service request	open
3	support assistant	resolve	service request	service request	open	service request	fixed
4	customer	approve	service request	service request	fixed	service request	closed
5	customer	reject	service request	service request	fixed	service request	open
6	customer	cancel	service request	service request (XOR)	submitted	service request	canceled

6	customer	cancel	service request	service request (XOR)	fixed	service request	canceled

Table 13. Structured representation of the customer service request handling BDD scenarios

Figures 10 – 13 show the diagrams created by the tool. Note that the layout of the activity diagram (Figure 13) differs from the activity diagrams shown in Section 6 regarding the symbols for the non-sequential control flow constructs. A merge node is shown as a × marker in the upper left corner of the activity of which the incoming control flows are merged (as in Figure 13). A join node is represented the same way but with a + marker. A fork node is shown as a + marker in the upper right corner of the activity of which the outgoing control flows are split. Also, start and end nodes are shown as lollypops that are sticked to activities, respectively on the left side and right side of these activities.

The use case diagram (Figure 10) shows user stories (as use cases) grouped per user role (as actor). This model provides a first graphical overview of the expected functionality of the system. The class diagram (Figure 11) emphasizes the service request object on which the system performs actions as requested by the agents in the different user roles. As service request is the only object on which actions are performed, the tool generates just one state machine diagram (Figure 12). This state machine diagram shows that a service request is created by a customer, and next moves through different states until it is cancelled or closed. The model shows when a certain action can be performed and what the result of this action is (in terms of a state change). For instance, a service request can be approved only if it is fixed. When the service request is approved, it is closed, and no further actions can be performed with it. Finally, the activity diagram (Figure 13) shows which user roles perform these actions and what actions may or must precede an action. For instance, the model shows that a customer can cancel a service request if that customer has created the service request and it has not been accepted by a support assistant or if the service request has been resolved by a support assistant but has not been approved by the customer.

Figure 10. Use case diagram generated from the structured representation of Table 13

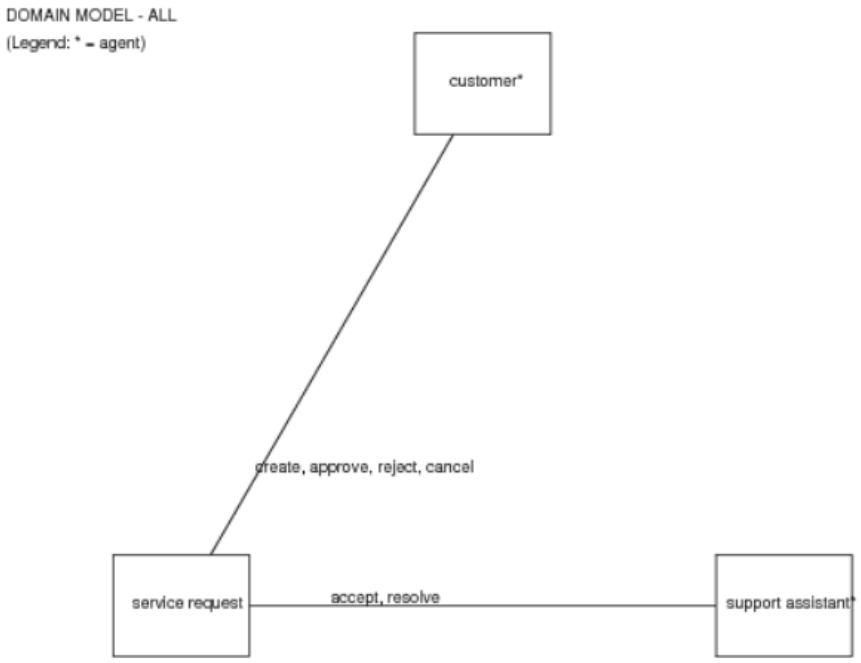

Figure 11. Class diagram generated from the structured representation of Table 13

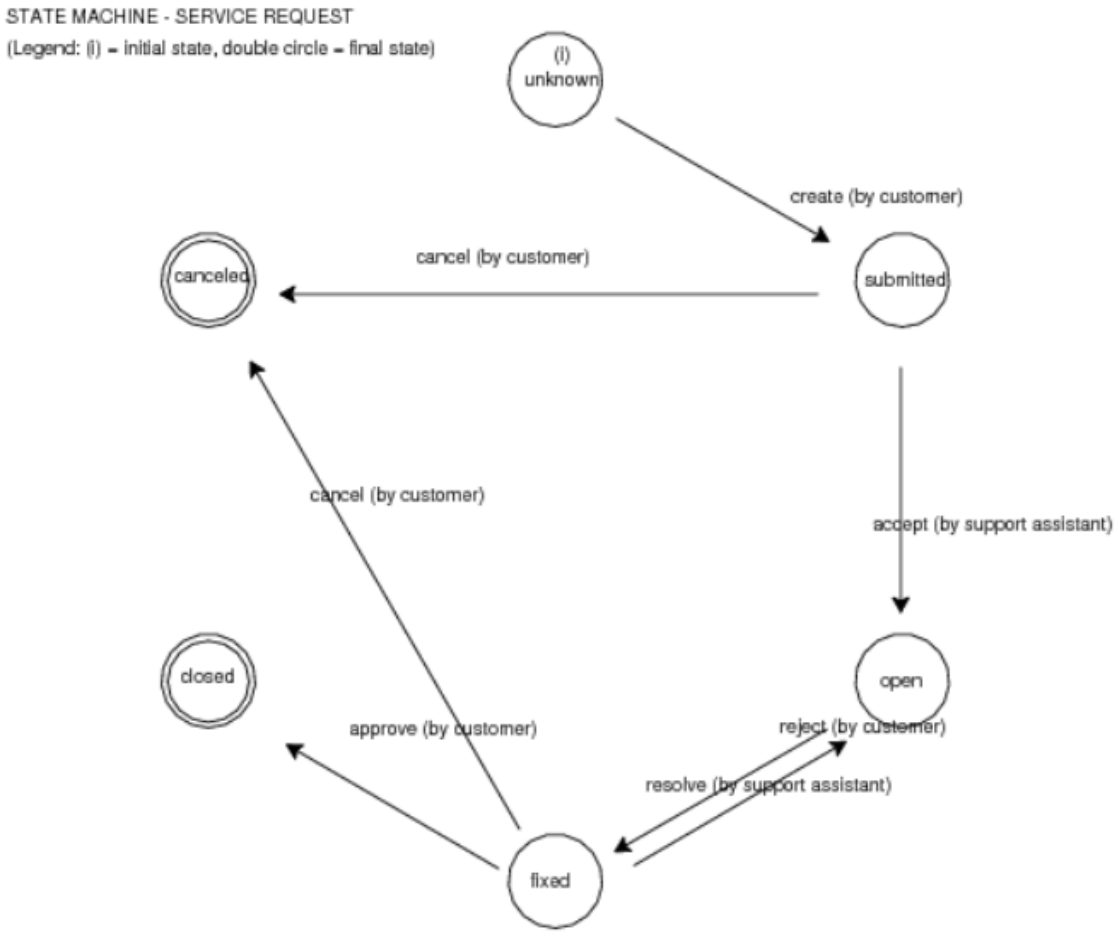

Figure 12. State machine diagram generated from the structured representation of Table 13

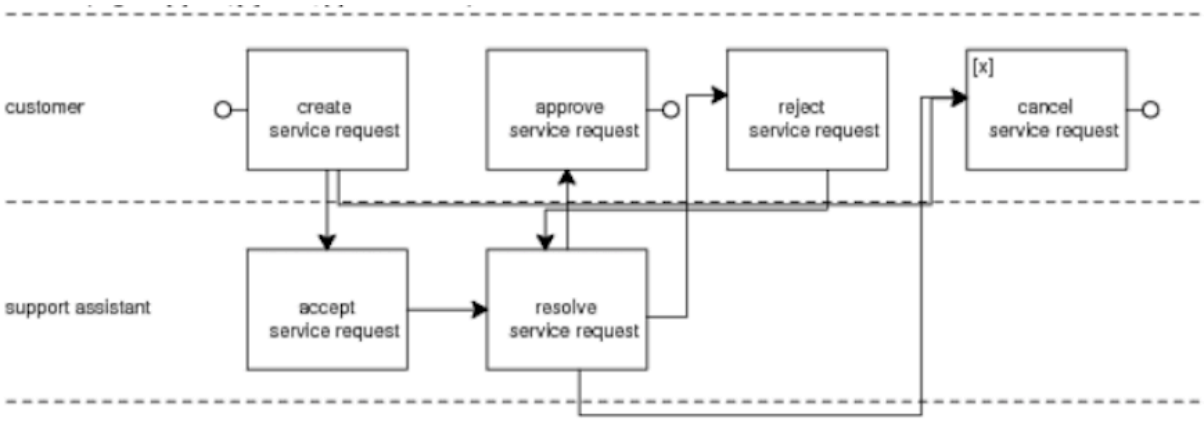

Figure 13. Activity diagram generated from the structured representation of Table 13

To illustrate how these models could be useful for requirements analysis, consider again BDD scenario 6:

> *As a customer, I want to cancel a service request so that the team can focus on other active requests. Given a service request is submitted or fixed, when the customer decides to cancel the service request, then the service request should be canceled.*

While this user story with its precondition, trigger and postcondition may express a perfectly valid requirement for the system in focus, the models may trigger questions related to the validity of this requirement. For instance, the activity diagram shows that the cancel service request activity by the customer may be preceded by the resolve service request activity by the support assistant. Also, the state machine diagram clearly shows that service requests may be cancelled when they are fixed. This means that after a support assistant resolved a service request, the customer can still decide to cancel it, which means that the effort to resolve the service request has been wasted. Again, this might be a valid requirement, but it could also signify an inaccuracy in the BDD scenario. The activity diagram shows that after the support assistant has resolved the service request, the customer can approve or reject it. So, should canceling the service request really be a third option?

Furthermore, the state machine diagram clearly shows that a service request cannot be cancelled in state open. Once a service request is open, the only next state allowed is fixed which means that a support assistant needs to spend effort to resolve the service request. This raises the question why a customer is not allowed to cancel a service request that is open. The state machine and activity diagrams further show that the open state of the service request may be the result of rejecting the service request after a support assistant has resolved it. This means that a support assistant again needs to spend effort on resolving the service request as it cannot be cancelled. Is this really a desirable situation? Or is the precondition of BDD scenario 6 inaccurate and should state fixed be state open? Or is the precondition incomplete, and should state open be added in disjunction to the submitted and fixed states?

To reason further, the use case diagram shows that only a customer can cancel a service request, which can also be inferred from the class diagram and activity diagram. This means that, hypothetically, a customer can continue rejecting service requests each time they have been resolved by a support assistant. The state machine diagram and activity diagram clearly show this loop in the behavior of the system. Currently, only the customer can break this loop by approving or canceling the service request. But maybe the requirements are incomplete, and a support assistant should be allowed to close an open service request to avoid endlessly spending effort on a service request that was submitted by an unsatisfiable customer? If that is the case, then a new BDD scenario may be added to the theme.

These illustrative situations show how the models generated from the BDD scenarios may help reasoning about the validity and quality of the requirements captured by the user stories and their preconditions, triggers, and postconditions. The models do not indicate that the requirements are inaccurate or incomplete per se, but they trigger questions that help to analyze these requirements. An analysis of the BDD scenarios themselves might of course also trigger such questions, but we posit that the graphical overview of the system's functionality and the different perspectives of this functionality that inheres in the selection of UML diagrams, makes the generated conceptual models useful when analyzing requirements. The next section evaluates whether this hypothesized usefulness is empirically supported.

8 Evaluation

To assess whether and how practitioners find the models generated by our tool useful for Agile software development, we conducted interviews with IT professionals. Given that we had a specific target group in mind (i.e., seasoned practitioners of Agile software development), we used a purposive sampling approach to select practitioners that could be considered experts in Agile software development. Table 14 shows the profiles of the eleven professionals we selected to participate in our study. These participants had an average of nineteen years of experience in IT jobs of which on average eleven years of Agile software development experience.

Participant ID	Years in IT	Years in Agile development	Current Job Title
F1	12	10	Senior Manager- Technology Transformation
F2	20	8	Application and Scrum Master
F3	21	11	Delivery Manager
F4	24	12	Delivery Lead
F5	24	12	Deputy Chief Microsoft Technology Associate
F6	20	18	Senior Manager- QA Delivery
F7	12	8	Delivery Lead
F8	16	10	Senior Project Manager
F9	30	14	Agile Transformation Coach
F10	15	11	Project Manager
F11	21	8	AVP Project Manager

Table 14. Participant profiles

We conducted semi-structured interviews with these participants via online Zoom sessions. These interviews were conducted in three stages. In the first stage, the participants were asked how conceptual models can help in Agile software development projects. In the second stage, a sample of BDD scenarios (the same six BDD scenarios as in Section 7) were shown and the four stylized UML diagrams were generated with the tool in front of the participants. The benefits of using conceptual models in Agile software development projects were then asked again. In the third stage, specific changes were made to BDD scenarios reflecting what could have been the outcome of requirements analysis activities (e.g., deleting a user story, changing a scenario). Participants were asked if they could detect those changes (which were applied without them seeing it). The revised BDD scenarios were then fed into the tool to regenerate the diagrams. Participants were shown both versions of the models (prior to change and after change) and were asked again the benefits of using conceptual models in Agile software development projects.

In the first stage of the interview, participants were not able to mention many uses of conceptual models in Agile software development. Nine out of eleven participants stated that they have not used models in projects employing Agile methods. However, the interviewees started mentioning potential general uses and benefits of the generated models in the second and third stages (Table 15). Other than the class diagram, all other models were perceived as useful.

Model type	**Perceived usefulness of the generated models**
Use Case Diagram	*to go back and review and relate the existing stories, brings visibility, to give the high representation of the solution, can get the total view of the entire solution, to identify the number of user stories, can help in writing test cases, can help in defining the scenarios, checking whether all scenarios are covered*
Activity Diagram	*Training, high-level planning like backlog grooming, can tell the flow and dependencies of the user stories, tells the entire process from end to end, can help to write end to end test scenarios, can help if it is a new initiative, can help in identifying the roles of people, develop navigational flow properly*
State Machine Diagram	*to verify the status of the stories that have been implemented, can help to visualize what are the missing stories or the new stories that should be added, can see if there are scenarios that they have to be covered while coding or testing, helpful for software enhancements, to identify the validation criteria and the criteria for exceptions*

Table 15: Uses and benefits of the generated models for Agile software development (verbatim from the interview transcripts)

From interview stage 2 onwards, some interviewees started reflecting specifically on the sample of BDD scenarios shown to them and mentioned specific uses and benefits of the generated models for supporting IT implementation (e.g., developing a software application for managing customer service requests). Verbatim transcripts of these uses and benefits are included in Table 16. These utterances demonstrate that the interviewees were getting engaged in considering the use of conceptual models in actual development, even if most of them never used conceptual models before in Agile software development.

Participant ID (time in the interview)	**Specific uses and benefits of the generated models for software development**
F1 (25:17)	*"yes, with this [use case diagram], I think it will help the developers to understand what kind of API they need to develop, or what kind of tables or the what kind of class or what kind of .net mode or what kind of stored procedures they can come up and write."*
F1 (33:56)	*"if I have this [state machine diagram] then I will be writing a get method, which will be able to get the details from the service request and then I'll be able to put it into a database so there will be a put method then once it goes by accept which will be another channel."*
F2 (18:59)	*"I think, if you talking about writing code and talking about different classes and sub classes and all those things will be useful from it [state machine diagram]. For example, when a request is submitted there'll be a submit class. When requests get cancel there will be a class called cancel and then, once the work is done and the work is accepted there will be accepted class which will be super class."*

F5 (20:30)	*"I like the state machine one because it tells me the validation criteria and the criteria for exceptions."*
F8 (7:45)	*"From the requirement traceability perspective and the requirement of integrity, these models is helpful."*
F9(24:04)	*"The end user can see how we are trying to implement this and they can provide feedback around this."*

Table 16: Specific uses and benefits of the generated models for software development

In stage 3 of the interview, after the experts were shown the modified conceptual models based on the changed BDD scenarios, many benefits of the automatic generation of models from BDD scenarios during requirements analysis were mentioned (Table 17). We learn from this that the interviewees clearly understood that models can be created automatically whenever there are changes in the BDD scenarios. The importance of writing clear user stories and scenarios, and how the model generation helps in this, was highlighted.

Participant ID (time in the interview)	**Benefits of automatically generating models from BDD scenarios during requirements analysis.**
F1 (42:17)	*"It actually automates the process rather than someone drawing it on their own. The architect can save the time."*
F2 (28:25)	*"User would use his own intelligence to question those specific keywords [that are changed in the user stories] so to write better requirements."*
F3 (20:33)	*"Definitely, the change in user stories will be much easier to identify. If I have made some mistake in the story like some validation that from 'close' it should not go to 'cancel' that will be easier to identify from this kind of diagram."*
F4 (24:32)	*"So one arrow [of the activity diagram] changes means not only just rework on that particular story that you change, it's also rework on the stories related to the fixed, closed, canceled and subsequently the testing of all the scenarios."*
F5 (26:08)	*"It could be an innocuous or as simple change, but can have a significant impact on the process and having a visual like this absolutely helps."*
F7 (24:54)	*"It will tell you what the word meant and how you missed that or misinterpreted it. That's a good thing, because a lot of the time we miss those small tiny things and then we end up delivering something else."*
F8 (22:51)	*"As you might take on additional stories and you know for future work, you can see, side by side, a as a current state model compared to what would be a future state model."*
F8 (23:04)	*"if you are doing any requirement change at that moment, you can, especially highlight here in the model the changes and you know that these changes are coming."*
F9 (22:03)	*"now that I have them side by side, I do see the model on the right, I do see that it is beneficial in the sense that it captures the flow properly."*
F10 (25:44)	*"automating it is definitely going to help us in generating all these models again and again that can be like many variations from our side. Product management side will be able to analyze all the variations."*

Table 17: Benefits of automatically generating models from BDD scenarios during requirements analysis

The main insight obtained from the interviews was that even if experts in Agile software development had never used conceptual models in Agile software development projects, they

perceived the usefulness of the set of models that we automatically generated from the sample set of BDD scenarios, which provides empirical support for the approach presented in this paper.

9 Discussion

9.1 Our Contribution to the State-of-The-Art

Generating conceptual models from user stories is not new. However, the related work that we reviewed generates, with few exceptions (e.g., Wautelet et al. [22]), just one type of model from user stories, often for a specific purpose or the use of the model is not specified. Furthermore, generated models fail to capture dependencies between user stories. The approach that we present in this paper is novel as we generate conceptual models for a set of related user stories (e.g., an epic or theme in Scrum) that is documented using BDD scenarios. These BDD scenarios do not only embed the actual user stories, but also specify their preconditions, triggers, and postconditions, and using this information we create models that allow understanding and analyzing dependencies between user stories.

With this approach, we contribute to the state-of-the-art in different ways. First, we define a consistent and complementary set of models to generate from BDD scenarios. We define these models as stylized versions of the following UML diagrams: use case diagram, class diagram, state machine diagram, and activity diagram. As far as we know, for state machine diagrams and activity diagrams (or other types of process model like BPMN diagrams) no approaches to automatically generate them from user stories have been proposed in the literature. While the class diagram and use case diagram provide a high-level overview of the system's functionalities and architecture, the state machine diagram and activity diagram allow analyzing dependencies between user stories. Together the four diagrams cover different perspectives useful for software engineering, including requirements analysis and model-driven software development.

Our second contribution is the design and implementation of an NLP-based text-to-model software solution that automatically generates the selected types of models when a set of BDD scenarios that document related user stories is inputted, given certain assumptions hold regarding the use of the defined BDD template. This solution is based on three different design artefacts: an agent-based framework to interpret a set of related user stories using a structured representation, concept mappings operationalized by mapping rules, and algorithms to implement those mapping rules along with a software tool that codes the algorithms. Using this tool, we demonstrated the feasibility of our approach, and we explored its use for requirements analysis.

The third contribution are the insights we provide on the usefulness of the models that we automatically generate from BDD scenarios. While the use of models in Agile software development is uncommon, an empirical study that we conducted with eleven expert practitioners of Agile software development unveiled many possible uses and benefits perceived by these experts, related to requirements analysis and software development in Agile software development projects. Even when most of the study participants were not using models in their practice, engaging with our tool inspired them to perceive usefulness in our approach.

9.2 Limitations

We discern three limitations to the model generation approach that we designed. A first limitation relates to the use of the theoretical framework based on agent-oriented modeling for interpreting the information contained in a set of BDD scenarios that document related user stories. Some agent-based concepts that could be related to BDD scenarios were not considered:

1. As entities, agents have states. The framework currently only considers objects to have states. Consequently, while both agents (i.e., user roles) and objects are modelled as classes in the class diagram, only for classes representing objects, state machine diagrams are created. As data storage and processing will focus on the objects, the classes representing the objects are the main architectural components of the system. However, modelling *roles* as agents, their state and behavior, can be interesting for requirements elicitation, system navigation and user interface design, and system security (e.g., authorization).
2. Agents pursue goals (i.e., proposition 1 of the Agent-based Framework (Table 3)). An agent-based concept such as goal could be used to interpret the *ends* part or answer to the 'why?' question of a user story. Also, the relationships between goals and objects (and their states) could be further defined to link *means* and *ends* parts of user stories, and possibly also the *outcome* part of BDD scenarios. As the generation of goal models from user stories has been investigated before (e.g., [17], [27], [22], [28]) and a goal model is not part of the selection of UML diagrams that our approach generates, the *ends* part of user stories, and thus BDD scenarios, was not considered in the concept mappings for the conceptual models.
3. Preconditions define when actions can be performed, but they do not trigger those actions. The framework could be extended with the concept of events as triggers of actions. The use of the *trigger* part of BDD scenarios for the creation of models as a basis for model-driven software development has recently been explored by Snoeck and Wautelet [40].

Second, we require that the user stories are formulated using the BDD scenario template of Table 1. This means that we impose a consistent level of granularity or detail on the writing of user stories (e.g., one-to-one mapping of user story and scenario). Also, we require the effect of actions to be described in terms of objects affected, without further detailing the states of these objects in terms of attributes. We further expect the standardized writing of a set of related user stories using this template, as described by our assumptions in sub-section 2.2. We recognize that these assumptions might not hold in practice, but how to deal with this is outside the scope of this paper. In other words, we focus on presenting a solution that works for user stories that are properly formulated using the BDD scenario template as we intend this template to be used. How to fix problems with improperly written user stories, is a relevant but pragmatic question that is independent of the conceptual solution presented in this paper.

Third, we had to resort to stylized versions of UML diagrams corresponding to the selection of a consistent and complementary set of conceptual models. The use of a simplified abstract syntax was needed as not all information usually found in these diagrams is captured by the proposed structured representation of a set of BDD scenarios that document related user stories, which also relates to the previous limitations. This limitation is harder to remove, however, for some aspects may not be impossible either. For instance, a lexical and semantic analysis of the wording of the different parts of BDD scenarios and comparison between related user stories, might reveal information that could be used to derive includes and extends relationships between use cases, attributes of objects described in user stories, multiplicities of the

relationships between such objects, specialization and aggregation relationships. The extent to which this will work without adding additional information to the BDD scenarios is subject to further research.

9.3 Future Research

Our future research is first aimed at improving the tool that we used in this paper for the proof-of-concept and empirical evaluation. At this stage, the tool is a working prototype that can generate the four types of stylized UML diagrams given that our assumptions hold regarding the BDD scenarios that are inputted. Using the tool, we showed the feasibility of our approach and engaged experts in Agile software development in reflecting on the perceived usefulness of automatically generated models in Agile software development projects. However, the diagrams outputted by the tool do not fully conform to UML concrete syntax. We are currently developing an export function that allows representing the generated models as XMI files which can then be imported in tools that better support UML. Further, we are investigating how the NLP-based techniques used by the tool can be further improved to make the tool more robust against deviations from our input assumptions (i.e., second limitation of sub-section 9.2) and how better use can be made of the information in BDD scenarios to bring the abstract syntax of the stylized UML diagrams closer to their standard UML counterparts (i.e., third limitation of sub-section 9.2). This investigation requires new iterations of the DSR process that we defined in Figure 1 as it also affects our other design artefacts.

We also plan to further explore use cases of conceptual models generated from BDD scenarios in Agile software development. An interesting avenue is the use of automatically generated models for model-driven software engineering and model-based code generation. In particular, the class diagram and state machine diagram seem to offer opportunities here [40]. We acknowledge that generating models is just a first step in making user stories 'executable', which is a challenge taken up in recent research in Agile software engineering [8, 9].

Finally, we are planning a more systematic empirical evaluation of the usefulness of those models generated by our tool that we believe are most helpful for requirements analysis in Agile software development projects: the use case diagram, activity diagram, and state machine diagram. To this end we are designing specific scenarios of validating and quality assuring requirements expressed by user stories (as BDD scenarios). In this scenario-based evaluation, we aim to compare the use of BDD scenarios with the joint use of BDD scenarios and automatically generated models to assess the added value offered by the models.

10 Conclusion

The research presented in this paper was guided by three questions. *What* is a set of models, rich in information about user stories and their dependencies, that is useful for Agile software development? *How* to generate these models? And *why* are these models useful, or stated differently, what is their use?

Using Design Science as research methodology, we designed, implemented, demonstrated, and evaluated an approach that automatically generates conceptual models from user stories. This approach generates simplified versions of UML class diagrams, use case diagrams, state machine diagrams, and activity diagrams, as these models are consistent and complementary in offering different software engineering perspectives on the system in development (i.e., the *what?* question). Our approach (i.e., the *how?* question) is based on different design artefacts. However, key to the solution is the use of BDD scenarios to document related user stories not only in terms of required functionality but also in terms of dependencies between those user

stories. As to the *why?* question, an empirical study with expert practitioners of Agile software development unveiled several uses and benefits of the automatically generated models, for requirements analysis, system design, software implementation, and testing. This perceived usefulness provides empirical support for the research we presented in this paper.

References

[1] I. Inayat and S. S. Salim, "A framework to study requirements-driven collaboration among agile teams: Findings from two case studies," *Computers in Human Behavior,* Article vol. 51, no. Part B, pp. 1367-1379, 10/1/October 2015 2015, doi: 10.1016/j.chb.2014.10.040.

[2] M. Cohn, *User Stories Applied: For Agile Software Development*. Boston: Addison-Wesley, 2004.

[3] D. Leffingwell, *Agile Software Requirements: lean requirements practices for teams, programs, and the enterprise* (Agile Software Development Series). Boston: Addision-Wesley, 2011.

[4] D. Leffingwell, *Safe 4.0 Reference Guide: Scaled Agile Framework for Lean Software and Systems Engineering*. Addison-Wesley Professional, 2017.

[5] B. Ramesh, C. Lan, and R. Baskerville, "Agile requirements engineering practices and challenges: an empirical study," *Information Systems Journal,* Article vol. 20, no. 5, pp. 449-480, 2010, doi: 10.1111/j.1365-2575.2007.00259.x.

[6] M. Daneva *et al.*, "Agile requirements prioritization in large-scale outsourced system projects: An empirical study," *The Journal of Systems & Software,* Article vol. 86, pp. 1333-1353, 5/1/May 2013 2013, doi: 10.1016/j.jss.2012.12.046.

[7] J. F. Smart, *BDD in action: Behavior-Driven development for the whole software lifecycle*. New York: Manning Publications Company, 2014.

[8] S. Heng, M. Snoeck, and K. Tsilionis, "Generating a Software Architecture out of User Stories and BDD Scenarios: Research Agenda," in *1st International Workshop on Agile Methods for Information Systems Engineering*, Leuven, Belgium, 2022, Leuven, Belgium: CEUR, pp. 40-46.

[9] K. Athiththan, S. Rovinsan, S. Sathveegan, N. Gunasekaran, K. S. A. W. Gunawardena, and D. Kasthurirathna, "An ontology-based approach to automate the software development process," in *IEEE Int. Conf. Inf. Automat. Sustainability (ICIAfS)*, Colombo, Sri Lanka, 2018, pp. 1-6, doi: 10. 1109/ICIAFS.2018.8913339.

[10] J. A. Hoffer, J. F. George, and J. S. Valacich, *Modern Systems analysis and design*, 6 ed. Pearson, 2011.

[11] Y. Wand and R. Weber, "Information Systems and Conceptual Modeling: A Research Agenda," *Information Systems Research,* vol. 13, no. 4, pp. 363-376, 2002.

[12] F. Bozyig, O. Aktas, and D. Kılınc, "Linking software requirements and conceptual models: A systematic literature review," *Engineering Science and Technology, an International Journal,* vol. 24, no. 1, pp. 71-82, 2021.

[13] B. Selic, "The pragmatics of model-driven development," *IEEE software,,* vol. 20, no. 5, pp. 19-25, 2003.

[14] V. N. Vithana, "Scrum Requirements Engineering Practices and Challenges in Offshore Software Development," *International Journal of Computer Applications,* vol. 116, no. 22, pp. 43-49, 2015.

[15] M. Trkman, J. Mendling, and M. Krisper, "Using business process models to better understand the dependencies among user stories," *Information and Software Technology,* Article vol. 71, pp. 58-76, 3/1/March 2016 2016, doi: 10.1016/j.infsof.2015.10.006.

[16] Y. Wautelet, S. Heng, D. Hintea, M. Kolp, and S. Poelmans, "Bridging User Story Sets with the Use Case Model," in *International Conference on Conceptual Modeling*, 2016, pp. 127–138, doi: 10.1007/978-3-319-47717-6.
[17] A. Jaqueira, M. Lucena, F. Alencar, C. M., and E. Aranha, "Using i* Models to Enrich User Stories," in *6th International i* Workshop*, 2013, vol. CEUR 978, pp. 55-60.
[18] M. Elallaoui, K. Nafil, and R. Touahni, "Automatic Transformation of User Stories into UML Use Case Diagrams using NLP Techniques," *Procedia Computer Science,* vol. 130, pp. 42-49, 2018, doi: https://doi.org/10.1016/j.procs.2018.04.010.
[19] F. Gilson, M. Galster, and F. Georis, "Generating use case scenarios from user stories,'' in Proc. Int. Conf. Softw. Syst. Processes, Jun. , pp. 31–40, doi: ," presented at the International Conference on Software and Systems Process, Seoul, 2020.
[20] I. K. Raharjana, D. Siahaan, and C. Fatichah, "User Stories and Natural Language Processing: A Systematic Literature Review," *IEEE Access,* vol. 9, pp. 53811-53826, 2021, doi: 10.1109/ACCESS.2021.3070606.
[21] T. Yue, L. C. Briand, and Y. Labiche, "aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models," *ACM Trans. Softw. Eng. Methodol.,* vol. 24, pp. 13:1-13:52, 2015.
[22] Y. Wautelet, S. Heng, and M. Kolp, "Perspectives on User Story Based Visual Transformations," in *22nd International Conference on Requirements Engineering: Foundation for Software Quality*, Utrecht, 2017, vol. 1796: CEUR.
[23] A. Azzazi, "A Framework using NLP to automatically convert User-Stories into Use Cases in Software Projects," *International Journal of Computer Science and Network Security,* vol. 17, pp. 71-76, 2017.
[24] T. Kochbati, S. Li, S. Gérard, and C. Mraidha, "From user stories to models: A machine learning empowered automation," in *9th Int. Conf. Model. Engineering Software Development*, 2021, pp. 28–40, doi: 10.5220/0010197800280040.
[25] S. Nasiri, Y. Rhazali, M. Lahmer, and N. Chenfour, "Towards a Generation of Class Diagram from User Stories in Agile Methods," in *International Workshop on the Advancements in Model Driven Engineering*, Warsaw, Poland, 2020, vol. 170: Procedia Computer Science, pp. 831-837, doi: 10.1016/j.procs.2020.03.148.
[26] W. Dahhane, A. Zeaaraoui, E. H. Ettifouri, and T. Bouchentouf, "An automated object-based approach to transforming requirements to class diagrams," in *2nd World Conf. Complex Syst. WCCS* 2014, pp. 158–163, doi: 10.1109/ICoCS.2014.7060906.
[27] T. Gunes and F. B. Aydemir, "Automated Goal Model Extraction from User Stories Using NLP," in *IEEE 28th International Requirements Engineering Conference (RE)*, 2020: IEEE.
[28] R. Mesquita, A. Jacqueira, C. Agra, M. Lucena, and F. Alencar, "US2StarTool: generation i* models from user stories.," in *International i* Workshop (iStar)*, 2015.
[29] G. Lucassen, M. Robeer, F. Dalpiaz, J. Van der Werf, and S. Brinkkemper, "Extracting conceptual models from user stories with Visual Narrator," *Requirements Engineering,* vol. 22, pp. 339-358, 2017.
[30] Y. Wautelet, S. Heng, S. Kiv, and M. Kolp, "User-story driven development of multi-agent systems: A process fragment for agile methods," *Computer Languages, Systems & Structures,* vol. 50, pp. 159-176, 2017.
[31] M. Bragilovski, F. Dalpiaz, and A. Sturm, "Guided Derivation of Conceptual Models from User Stories: A Controlled Experiment," in *In International Working Conference on Requirements Engineering: Foundation for Software Quality*, Birmingham, 2022, vol. 13216: Springer-Verlag, pp. 131-147.

[32] A. Hevner, S. March, J. Park, and S. Ram, "Design Science Research in Information Systems," *MIS Quarterly,* vol. 28, no. 1, pp. 75-105, 2004.
[33] S. Gregor and A. Hevner, "Positioning and presenting design science research for maximum impact," *MIS quarterly,* vol. 37, no. 2, pp. 337-355, 2013.
[34] G. Lucassen, F. Dalpiaz, J. Van der Werf, and S. Brinkkemper, "The use and effectiveness of user stories in practice," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, 2016, pp. 205-222.
[35] Axelos, *ITIL Foundation: ITIL 4*. London, UK: The Stationery Office, 2019.
[36] F. Dalpiaz, P. Gieske, and A. Sturm, "On deriving conceptual models from user requirements: An empirical study," *Information and Software Technology,* vol. 131, 2021.
[37] M. Menveld, S. Brinkkemper, and F. Dalpiaz, "User Story Writing in Crowd Requirements Engineering: The Case of a Web Application for Sports Tournament Planning," in *IEEE 27th International Requirements Engineering Conference Workshop*, 2019.
[38] L. Binamungu, S. Embury, and N. Konstantinou, "Characterising the Quality of Behaviour Driven Development Specifications. In Agile Processes in Software Engineering and Extreme Programming," in *21st International Conference on Agile Software Development*, Copenhagen, Denmark, 2020, vol. 383: XP 2020, pp. 87-102.
[39] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems,* vol. 24, no. 3, pp. 45-77, 2007.
[40] M. Snoeck and Y. Wautelet, "Agile MERODE: A Model-driven Software Engineering Method for User-centric and Value-based Development," *Software and Systems Modeling,* pp. 1-26, 2022.
[41] I. Jacobson, *Object-oriented software engineering : a use case driven approach*. Reading, Mass: Addison-Wesley, 1992.
[42] K. E. Kendall and J. E. Kendall, *Systems Analysis and Design*, 10 ed. Pearson, 2019.
[43] K. Markham, Mintzes, J., Jones, M., "The concept map as a research and evaluation tool: Further evidence of validity," *Journal of Research in Science & Teaching,* vol. 31, no. 1, pp. 91-101, 1994.
[44] S. White, "Business Process Modeling Notation." Accessed on: 04/06/2022
[45] J. Mendling and J. Recker, "Towards systematic usage of labels and icons in business process models," in *12th International Workshop on Exploring Modeling Methods in Systems Analysis and Design*, CEUR, Montpellier, France, 2008, pp. 1-13.
[46] A. Dennis, B. H. Wixom, and D. Tegarden, A. Dennis, Ed. *Systems Analysis and Design: An Object-Oriented Approach with UML*, 5 ed. Wiley, 2015.
[47] M. Robeer, G. Lucassen, J. Van der Werf, F. Dalpiaz, and S. Brinkkemper, "Automated extraction of conceptual models from user stories via NLP," IEEE 24th International Requirements Engineering Conference (RE), 2016, pp. 196-205.
[48] K. Tsilionis, J. Maene, S. Heng, Y. Wautelet, and S. Poelmans, "Conceptual Modeling Versus User Story Mapping: Which is the Best Approach to Agile Requirements Engineering?," in *Research Challenges in Information Science: 15th International Conference*, Limassol, Cyprus, S. S. Cherfi, A. Perini, and S. Nurcan, Eds., 2021, vol. 415: Springer, Lecture Notes in Business Information Processing, pp. 356–373.
[49] Y. Wautelet, S. Heng, M. Kolp, and I. Mirbel, "Unifying and Extending User Story Models," in *Conference on Advanced Information Systems Engineering*, Thessaloniki, Greece, M. Jarke *et al.*, Eds., 2014, vol. 8484: Lecture Notes in Computer Science, pp. 211–225.

[50] E. Yu, "Agent-Oriented Modelling: Software versus the World," in *Agent-Oriented Software Engineering*, Berlin, Heidelberg, W. M.J., W. G., and C. P, Eds., 2002, vol. 2222: Lecture Notes in Computer Science, doi: https://doi.org/10.1007/3-540-70657-7_14.
[51] C. Castelfranchi, "Modelling social actions for AI agents," *Artificial Intelligence,* vol. 103, pp. 157-182, 1998.
[52] S. Franklin, *Artificial Minds*. Cambridge, MA: MIT Press, 1995.
[53] M. Wooldridge, *Reasoning about Rational Agents*. Massachusetts: The MIT Press, 2000.

Appendix A – Pseudo-code algorithms to generate conceptual models from the structured representation of a set of BDD scenarios that document related user stories (e.g., taken from an epic or theme in Scrum)

Exhibit A2 presents the algorithm that implements mapping rules 1.1 to 1.3 that generate a use case diagram from the structured representation of a set of BDD scenarios that document related user stories.

```
Input: Structured representation of a set of BDD scenarios that document related user stories
Output: Use case diagram
1:    for each row in the structured representation
2:        if the agent is not already represented as an actor then
3:            create a new actor
4:        end if
5:        if the action – object is not already represented as a use case then
6:            create a new use case
7:        end if
8:        if the actor representing the agent and the use case representing the action - object are not already connected by an association then
9:            create an association between the actor and the use case
10:       end if
11:   end for
```

Exhibit A3. Algorithm to create a use case diagram

Exhibit A2 presents the algorithm that implements mapping rules 2.1 to 2.3 that generate a class diagram from the structured representation of a set of BDD scenarios that document related user stories.

```
Input: Structured representation of a set of BDD scenarios that document related user stories
Output: class diagram
1:        for each row in the structured representation
2:            if the agent is not already represented as a class then
3:                create a new class
4:            end if
5:            if the object is not already represented as a class then
6:                create a new class
7:            end if
8:            if the class representing the agent and the class representing the object are not already connected by an association representing the action then
9:                create an association between the two classes and label the association with the name of the action
10:           end if
11:       end for
```

Exhibit A2. Algorithm to create a class diagram

The mapping rules for state machine diagrams (3.1 to 3.6) are implemented in the algorithm presented in Exhibit A3. We annotated the different segments of this algorithm with the identifying numbers of the mapping rules to increase transparancy.

Input: Structured representation of a set of BDD scenarios that document related user stories
Output: A set of state machine diagrams

```
1:  for each row in the structured representation
2:      if the object on which the action is performed is not already
        represented as a state machine diagram then
3:          create a new state machine diagram for this object
            (confer mapping rule 3.1)
4:      end if
5:  end for
6:  for each row i in the structured representation
7:      if for the precondition object a state machine diagram was created
        then
8:          if the precondition state is not already represented as a state in
            that state machine diagram then
9:              create a new state in the state machine diagram for the
                object
                (confer mapping rule 3.2)
10:         end if
11:         set initial state to true
12:         for each row j in the structured representation (action row i ≠
            action row j)
13:             if postcondition object row j = precondition object row i
                AND postcondition state in row j = precondition state
                in row i then
14:                 set initial state to false
15:             end if
16:         end for
17:         if initial state is true then
18:             indicate that the state representing the precondition
                state in the state machine diagram for the object is an
                initial state
                (confer mapping rule 3.4)
19:         end if
20:     end if
21:     if for the postcondition object a state machine diagram was created
        then
22:         if the postcondition state is not already represented as a state in
            that state machine diagram then
23:             create a new state in the state machine diagram for the
                object
                (confer mapping rule 3.2)
24:         end if
25:         set final state to true
26:         for each row j in the structured representation (action row i ≠
            action row j)
27:             if precondition object row j = postcondition object row i
                AND precondition state in row j = postcondition state
                in row i AND postcondition object in row j =
                postcondition object in row i then
```

```
28:                                     set final state to false
29:                                 end if
30:                             end for
31:                             if final state is true then
32:                                 indicate that the state representing the postcondition
                                    state in the state machine diagram for the object is a
                                    final state
                                    (confer mapping rule 3.5)
33:                             end if
34:                         end if
35:                         if precondition object = postcondition object AND postcondition state
                            ≠ precondition state then
                                create in the state machine diagram for the object, a transition
                                from the state that represents the precondition state to the state
                                that represents the postcondition state and label the transition
                                with the name of the action
                                (confer mapping rule 3.3)
36:                         end if
37:                         set not in precondition to true
38:                         for each row j in the structured representation (action row i = action
                            row j)
39:                             if postcondition object row i = precondition object row j then
40:                                 set not in precondition to false
41:                             end if
42:                         end for
43:                         if not in precondition is true then
44:                             create in the state machine diagram for the object, a state
                                labelled 'unknown' and create a transition from this 'unknown'
                                state to the state that represents the postcondition state and
                                label the transition with the name of the action
                                (confer mapping rule 3.6)
45:                         end if
46:                 end for
```

Exhibit A3. Algorithm to create state machine diagrams

The algorithm in Exhibit A4 implements mapping rules 4.1 and 4.2 to create the Stage 1 activity diagram.

```
Input: Structured representation of a set of BDD scenarios that document related user
stories
Output: A partial activity diagram (Stage 1)
1:  for each row in the structured representation
2:      if the agent is not already represented as a swimlane then
3:          create a new swimlane for the agent
4:      end if
5:      if the action-object pair is not already represented as an activity then
6:          create a new activity for the action-object pair in the swimlane
            representing the agent performing the action on the object
7:      end if
8:  end for
```

Exhibit A5. Algorithm to create the activity diagram (Stage 1)

The algorithm is continued in Exhibit A6 with the implementation of mapping rule 4.3 to create the Stage 2 activity diagram.

Input: Structured representation of a set of BDD scenarios that document related user stories & Stage 1 activity diagram generated from that structured representation
Output: A partial activity diagram (Stage 2)

```
9:    for each row i in the structured representation
10:       for each row j in the structured representation (action row i ≠ action row j)
11:          if postcondition object of row i = precondition object of row j AND
             postcondition state of row i = precondition state of row j then
12:             draw a control flow from the activity representing the action-
                object pair in row i to the activity representing the action-
                object pair in row j
13:          end if
14:       end for
15:   end for
```

Exhibit A7. Algorithm to create the activity diagram (Stage 2)

For implementing mapping rules 4.4 and 4.5, the algorithm is further extended as shown in Exhibit A8. The algorithm now creates a Stage 3 activity diagram.

Input: Structured representation of a set of BDD scenarios that document related user stories & Stage 2 activity diagram generated from that structured representation
Output: A partial activity diagram (Stage 3)

```
16:   for each row i in the structured representation
17:       set unmatched precondition to true
18:       set unmatched postcondition to true
19:       for each row j in the structured representation (action row i ≠ action row j)
20:          if precondition object of row i = postcondition object of row j AND
             precondition state of row i = postcondition state of row j then
21:             set unmatched precondition to false
22:          end if
23:          if postcondition object of row i = precondition object of row j AND
             postcondition state of row i = precondition state of row j then
24:             set unmatched postcondition to false
25:          end if
26:       end for
27:       if unmatched precondition is true AND no start node has already been
          connected to the activity representing the action-object pair in row i then
28           create a start node and draw a control flow from the start node to the
             activity representing the action-object pair in row i
29:       end if
30:       if unmatched postcondition is true AND no end node has already been
          connected to the activity representing the action-object pair in row i then
31:          create an end node and draw a control flow to the end node from the
             activity representing the action-object pair in row i
32:       end if
33:   end for
```

Exhibit A9. Algorithm to create the activity diagram (Stage 3)

Finally, the algorithm is completed in Exhibit A10 with the implementation of mapping rules 4.6 to 4.8 to create the Stage 4 (i.e., final) activity diagram.

Input: Structured representation of a set of BDD scenarios that document related user stories & Stage 4 activity diagram generated from that structured representation
Output: An activity diagram (Stage 4)

```
34:  for each row i in the structured representation
35:      if an XOR logical operator was added to the precondition object then
36:          create a merge node in the activity diagram
37:          connect all control flows directed to the activity that represents the
             action of row i, to this merge node as incoming control flows
38:          create a control flow going out of the merge node and direct it to the
             activity that represents the action of row i
39:      end if
40:      Skip all rows j (action row i = action row j)
41:  end for
42:  for each row i in the structured representation
43:      if an AND logical operator was added to the precondition object then
45:          create a join node in the activity diagram
46:          connect all control flows directed to the activity that represents the
             action of row i, to this join node as incoming control flows
47:          create a control flow going out of the join node and direct it to the
             activity that represents the action of row i
48:      end if
49:      Skip all rows j (action row i = action row j)
50:  end for
51:  for each row i in the structured representation
52:      if an AND logical operator was added to the postcondition object then
53:          create a fork node in the activity diagram
54:          connect all control flows leaving from the activity that represents the
             action of row i, to this fork node as outgoing control flows
55:          create a control flow from the activity that represents the action of
             row i, to the fork node
56:      end if
57:      Skip all rows j (action row i = action row j)
58:  end for
```

Exhibit A11. Algorithm to create the activity diagram (Stage 4)