Architecture for Incorporating Internet-of-Things Sensors and Actuators into Robot Task Planning in Dynamic Environments

Helen Harman, Keshav Chintamani and Pieter Simoens
Department of Information Technology - IDLab
Ghent University - imec
Technologiepark 15, B-9052 Ghent, Belgium
{firstname.surname}@ugent.be

Abstract— **Robots are being deployed in a wide range of smart environments that are equipped with sensors and actuators. These devices can provide valuable information beyond the perception range of a robot's on-board sensors, or provide additional actuators that can complement the robot's actuation abilities. Traditional robot task planners do not take these additional sensor and actuators abilities into account. This paper introduces an enhanced robotic planning framework which improves robots' ability to operate in dynamically changing environments. To keep planning time short, the amount of knowledge in the planner's world model is minimized.**

Index Terms— **robotics, IoT, smart environments, task planning**

I. INTRODUCTION

The deployment of mobile robots is being envisioned by the research community in an increasing number of environments for long-term autonomy, including assistance for elderly at home, and service robots in warehouses. These robots work alongside humans, therefore must adapt to dynamic changes in the environment.

Instead of hard coding sequences of actions to execute a given task, a more generic approach is to use symbolic task planning, an artificial intelligence technique that is gaining traction in real-world robotics. In symbolic task planning, a task is formulated as a desired goal state, and planners autonomously find the appropriate set of actions (i.e. a task plan) to move from the current state to the desired goal state. In dynamic real-world environments, task plans are adjusted continuously, according to new observations by the robot's sensors that are mapped to state updates.

An increasing number of spaces are being equipped with network-enabled sensors and actuators, such as cameras, presence detectors, lifts, locks and robots. Together with back-end cloud services for data processing, this Internet-of-Things (IoT) can be clustered behind what we define in this paper as a smart environment. This provides additional sensing and actuation abilities that can aid robots in achieving their goals.

In this paper, we propose an on-robot planning paradigm that leverages on a smart environment. The robot will be able to adapt its plan according to the state of the environment provided by both its on-board sensors and remote IoT sensors. As an example scenario, consider a robot instructed to fetch a coffee in a smart office environment. An IoT coffee machine may inform the robot that it is empty, or a door sensor may indicate that a door is closed on the route towards the coffee machine. The robot can use this knowledge to pre-empt its current plan and identify an alternative coffee machine. Without the IoT sensor information, the robot would only find out about the empty coffee machine or closed door when in front of it.

As well as providing up-to-date information on the environment and its dynamics, a smart environment can aid a robot by performing actuations which the robot itself is incapable of. The task planner can include these off-board actuation capabilities in its plan. Continuing with our example, the robot is incapable of making a coffee so requests the IoT coffee machine to do this, or the robot may be unable to open a closed door because it is holding a cup. Without the aid of these smart environment actuators the robot would be unable to accept or complete a coffee fetching task.

Through incorporating sensors and actuators from smart environments into robot task planning, we expand the scope of symbolic task planners beyond the action capabilities and sensor perception range of the robot itself. Our system improves a robot's ability to autonomously operate in dynamically changing environments and increases the scope of tasks that can be executed.

In section II we review related work. Section III gives an overview of our system; this is expanded on in section IV which provides further details on the implementation.

II. RELATED WORK

In this section we introduce some of the existing approaches to robot task planning, in particular we focus on those which attempt to handle dynamic environments, we also provide examples of how robots have been incorporated into smart environments. What aspects of these we aim to improve upon will then be discussed.

A. Task Planning in Dynamic Environments

Symbolic planners use a solver (e.g. STRIPS [1], TFD/M [2]) to find a set of actions to transform the initial state to a goal state. The Problem Domain Definition Language (PDDL) is a popular domain-independent logic-based formalism to represent planning problems.

PDDL represents the world and actions in a problem and domain file respectively. The problem file contains an initial (current) state and a goal state, both expressed using predicates, fluents and objects. Actions, along with their durations, conditions and effects are listed in a domain file.

In real-world open environments, it is likely that the state given to the planner is either incomplete or out-of-date. One approach is to generate plans with conditions: which branch of the plan is executed depends on sensor observations at runtime. This approach is followed by the Planning With Knowledge and Sensing framework [3], e.g. the robot senses at runtime whether a beverage container is filled or not and will manipulate the container appropriately.

Alternatively, a replanning procedure can be carried out if there is a mismatch between the state belief from which the plan was generated and the actually sensed world state. Continuous planners iterate between sensing, planning and acting. Once an action has finished, be it successful or otherwise, the planner's current state is updated. If the preconditions of the next planned action are no longer met, replanning is invoked. Several architectures have been proposed to transform the robot sensor observations into state updates. In [4], sensor information is used to update an ontology that is queried in each planning loop to populate a PDDL problem file. In [5], the PDDL is updated from several state estimator plugins.

In the above works, information from remote sensors is not used and the planner can only search through actions on the robot itself.

B. *Task Planning in smart environments*

Smart environments enable robots to gain knowledge from external sources, including the cloud and IoT devices (e.g. smart coffee machines, door sensors and other robots). In the RoboEarth project [6] the cloud populates a robot's knowledge-base, which includes information on action recipes and objects found in the environment. Once a robot has executed its plan it uploads semantic maps containing object locations, allowing them to be shared with other robots.

IoT devices can provide up-to-date information on the actual environment state and its dynamics, in turn helping the robot to execute plans more efficiently. [7] introduces the concept of a robot ecology, consisting of a collection of physically embedded intelligent systems distributed in a smart environment. For each action in a task plan, a centralized configuration planner sets up the appropriate communication channels between the distributed systems, e.g. to send the observations of a door-mounted camera to a robot crossing that door. This configuration planner is compatible with our approach and can be used in the smart environment to communicate the most appropriate sensors for a given task to the robot. Our approach provides additional support for plan pre-emption by the smart environment and does not work on a predefined list of all available actions but instead only sends relevant actions to the task planner.

III. System Concept

A key consideration in the system design is the distribution of the task planner components between the robot and a remote (cloud) server in the smart environment. Running the task planner on the cloud has the advantage that it is deployed close to where IoT sensor data is processed. As our goal is to improve the long-term autonomy of mobile robots, we have opted to execute the task planner on the robot allowing it to continue operating when wireless network connectivity drops. However, the amount of information transmitted from the smart environment to update the robot's world model must be limited for two reasons. First, wireless communication is energy hungry and transmitting raw IoT sensor data streams would quickly drain the robot's battery. Second, introducing all IoT sensors and actuators as objects in the planner's world model increases the model's complexity causing the planning time to increase exponentially [8], [9]. We thus aim to keep the world model compact by only including relevant actions and filtering out objects which have no impact on the task plan.

A functional overview of our continuous planning system is presented in Figure 1.

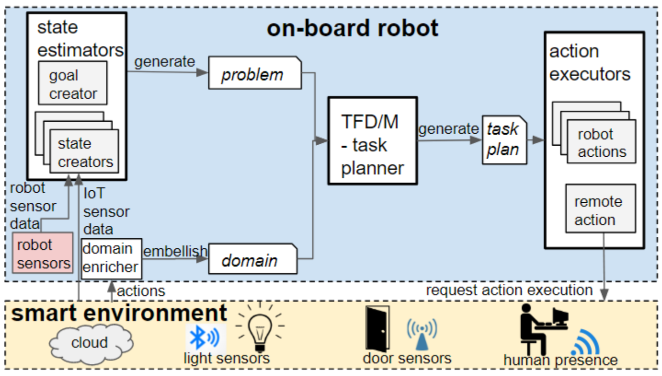


Fig. 1. The continual planner calls the state estimators (i.e. goal creator and state creators) and domain enricher to generate the PDDL problem and domain based on state observations from the robot and from the smart environment. The resulting task plan contains actions to be executed by the robot and other actuators in the smart environment.

Our system is an expanded version of the continuous planning framework of Dornhege et al. [5]. This framework provides two types of interfaces between the domain-independent TFD/M solver and a real-world system: state estimators and action executors. In the original framework, state estimator plug-ins translate robot sensor observations into PDDL state updates, while action executor plug-ins translate PDDL-defined actions into executable robot instructions. Below, we explain how we have modified and expanded this framework.

A. *Problem Generation*

The PDDL problem file contains a goal and the current state and is populated by calling a set of state estimator plug-ins. The goal creator is called once when the continuous planner is started, and state creators are called at the start of each sense-plan-act iteration. The planning step is only invoked when the conditions for the next planned action are not met.

We allow state creators to update the problem using both robot sensors and IoT sensors. Examples of robot sensed state include robot position determined by odometry, blocked locations determined using the robot's path planner and occupancy map; and objects identified in the robot camera view. Using IoT sensed state expands the robot's knowledge beyond the scope of its own sensors. One example is an obstacle detected in a hallway via a CCTV camera, or the open/closed status reported by a door sensor.

B. *Domain enricher*

The PDDL domain file contains (a.o.) the set of actions from which the planner generates a task plan. In traditional approaches, this list was limited to a fixed set of actions. In realistic environments, this list can quickly grow in size: e.g. for each object that a robot may encounter a different manipulation action could be defined.

Instead of always using an exhaustive predefined list of actions in the planner, we start with a minimal domain file only containing the most elementary actions a mobile robot can execute: navigate the environment and inspect objects. A *domain enricher* process analyses the PDDL problem obtained after calling the state creators and only adds actions that are relevant to entities defined. For example, if there are manipulatable objects in the problem file, the domain enricher will include grasping actions in the domain file.

The domain enricher can leverage on knowledge databases in the cloud to determine available actions. For example, one might use ontology reasoning to determine feasible actions in the given context. In our current implementation, we use a static repository that is queried by the domain enricher.

As we include actions that should be performed by other actuators in the smart environment, we add a single remote device object to the PDDL problem in line with our spirit to keep solver times tractable. Actions defined in the domain contain a condition stating if it should be executed locally or remotely.

C. *Action Executors*

Our system enables the generation of task plans containing a mixture of robotic actions and actions that are delegated to the smart environment, e.g. to open an actuated door. Action executors are plug-ins that contain the logic to execute an action defined in the PDDL domain file. Different from the original action executor plug-ins in [5], we do not allow PDDL state updates to happen from within action executors. Instead, we delegate such functionality to the state estimators described earlier, decoupled from action execution. Any remote action in the task plan is delegated to a single action executor plug-in that acts as a proxy for all off-board actuators in the smart environment. When remote action execution is required the action request is sent to the cloud where it is redirected to the appropriate actuator service, possibly using an IoT middleware solution to abstract from vendor-specific syntax.

IV. Implementation

So far we have presented a broad overview of the system, therefore further details will be given in this section. In section IV-A, we present the implementation of our architecture. In section IV-B, we provide insight in the state creators and action executors we have developed for the basic scenario of a mobile robot navigating in a smart environment.

A. *ROS-based planning framework*

We have implemented our planning framework in the Robot Operating System (ROS) [10], allowing us to extend existing ROS packages for PDDL based planners such as `tfd_modules`. ROS applications run inside of nodes, individual processes which communicate by publishing/subscribing to topics and providing/invoking services. In our architecture (see Fig. 2) all ROS nodes run on-board the robot, as the robot should remain operational when no remote connection is available. The cloud aspect of our system does not use ROS in order to keep it decoupled from the robotic middleware.

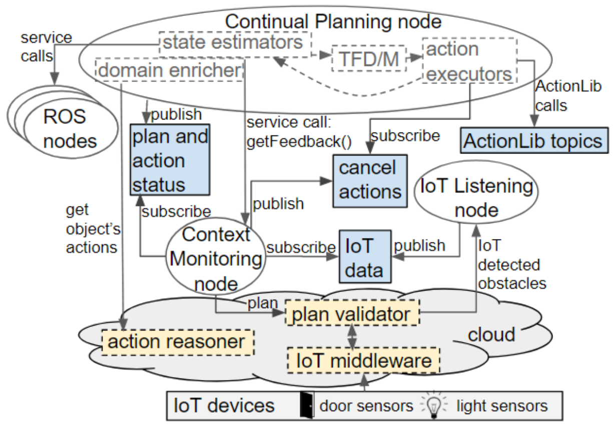


Fig. 2. Ovals represent ROS nodes and boxes with a blue background show ROS topics. All communication to/from the cloud is performed using HTTP requests.

The Continual Planning node loops through the phases of populating a PDDL problem and domain file using state estimators and the domain enricher; running the TFD/M solver, and then calling the appropriate action executor plug-ins which interface with actuator drivers through the ROS ActionLib interface.

The Context Monitoring node plays a central role. State estimators may query this node to get information about what actions have been executed, the current plan, and to get relevant IoT sensor data. For example, it is used by state creators to discover when an action has failed; and to determine what objects should be present in the problem.

The Context Monitoring node also announces the robot's plan to the cloud, where a plan validator performs reasoning on which sensors in the smart environment could provide possible relevant sensor input to the cloud. In turn, the cloud will push relevant sensor observations to a robot's IoT Listening node.

B. *State creator and actuation executor plug-ins*

We have developed state estimators and actuation executor plug-ins for the basic scenario of a mobile robot fetching objects in a multi-room environment, listed in Table I and Table II. While navigating, the robot may encounter obstacles such as boxes or closed doors. The robot is able to push boxes out of the way, but it cannot open doors. All doors are equipped with a sensor, but only a few doors in the environment are equipped with an electronic opener.

The goal_creator adds the waypoints of the environment, and the target position in the problem; and the robot_pose state creator adds the robot's current location. Only the drive-base (shown in Listing 1) and inspect-object actions are defined upfront in the PDDL domain. When obstacles on the robot's path are detected, its state is updated and replanning is triggered. At this stage, other state estimators and action executor plug-ins come into play.

```
(:durative-action drive-base
 :parameters (?r - robot ?s - location ?g -
     ↪location)
 :duration (= ?duration 1000)
 :condition (and
   (at start(at-base ?s ?r))
   (at start(not(at-base ?g ?r)))
   (over all(is-local ?r))
   (over all(can-move-to ?s ?g))
 )
 :effect (and
   (at start(not(at-base ?s ?r)))
   (at end(at-base ?g ?r))
 )
)
```

Listing 1. drive-base action from the PDDL domain. can-move-to is a derived rule which checks the path can be traversed (i.e. locations are in the same room or in-line, and no objects are between them).

To illustrate this, we present two different situations in the simulated world shown in Figure 3. In this figure and all the next ones, the numbers are the room IDs, the text indicates different locations which are abbreviated (e.g d1_r1 is doorway1_room1 and b1 is blocked_loc1) and the arrows represent the drive-base actions. A robot can navigate between locations if they are in the same room or are either side of a doorway.

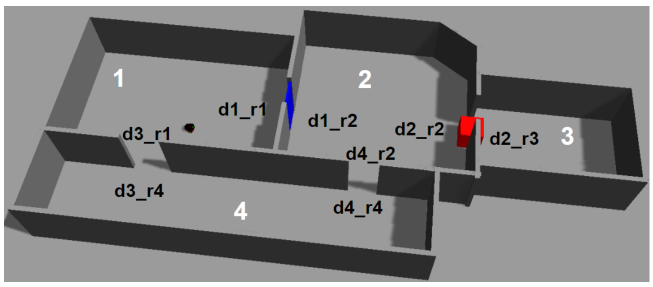
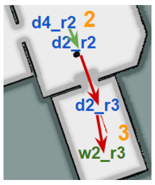


Fig. 3. Simulated world using Gazebo. door1 is shown in blue, box1 is the red box. The blue text shown the different locations a robot can navigate to. This is an expanded version of the simulated world created by Speck et al. [11].

In the first situation, the robot detects via its on-board sensors that a box is blocking its path. In the second situation, it is the smart environment that pro-actively detects that a door further along the robot's path was just closed. While these situations are elementary, they effectively demonstrate the interplay between state estimators, domain enricher and smart environment.

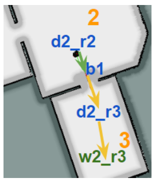
1) Obstacle detected by the robot: This scenario is illustrated in Figure 4. The robot is currently in room 2 and is asked to be in room 3. As the robot has no knowledge on obstacles, the initial plan contains three drive-base actions. Because an obstacle is blocking the doorway between both rooms, the second drive action will fail and the robot's state estimation and replanning are triggered.

(A)



1. (drive-base r1 d4_r2 d2_r2)
2. (drive-base r1 d2_r2 d2_r3) ← action fails causing replanning
3. (drive-base r1 d2_r3 w2_r3)

(B)

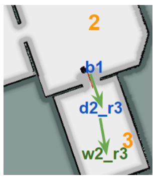


1. (drive-base r1 d2_r2 b1)
2. (inspect-object r1 d2_r2 b1)

State change causes replanning

3. (drive-base r1 b1 d2_r3)
4. (drive-base r1 d2_r3 w2_r3)

(C)



1. (push-box r1 b1 d2_r3 box0)
2. (drive-base r1 b1 d2_r3)
3. (drive-base r1 d2_r3 w2_r3)

Fig. 4. Rviz displaying the robot's costmap, location and goal, alongside the task plans used when an obstacle (box1) is detected by on-board sensors. The robot's initial location is doorway4_room2 and has been assigned the goal: (at-base waypoint2_room3 robot1).

When the blocked_locations state estimator is called, it uses the report of the failed drive action and the path planner to add an additional location, which indicates the position of the unknown object, to the PDDL problem, as shown in Listing 2. The new plan is shown in Fig. 4-B and now contains four actions, including an inspect-object action.

```
(:objects  blocked_loc1 - location)
(:init
 (is-blocked blocked_loc1 doorway2_room3)
 (is-unknown-object-loc blocked_loc1)
 (= (x blocked_loc1) 11.0)
 (= (y blocked_loc1) 1.0)
 (= (z blocked_loc1) 0)
)
```

Listing 2. PDDL problem showing a blocked location. is-blocked indicates that the robot can not drive between the two locations; and is-unknown-object-loc indicates that that robot detected the obstacle from the blocked_loc1 location.

The robot starts executing the new plan, it drives to the obstacle and inspects it. The inspect_object action executor performs image recognition on the robot's RGB camera feed and publishes that it discovered a box[1]. The result of the

[1]Note that this image recognition is possibly performed in the cloud, but this is not relevant for our current discussion.

TABLE I

DESCRIPTION OF THE DIFFERENT STATE ESTIMATORS

State estimator	Description
goal_creator	Adds the PDDL goal string and static information to the problem. Waypoints, which are listed in a text file and written in the format <*ID*>_<*roomID*>, are added and those with matching IDs (e.g. locations either side of doorways) are set as being in-line. Based on Speck et al.'s work [11].
robot_pose	Obtains the robot position from odometry. If the position is equivalent to a location that has previously been added to the state, the robot is assigned to the location using the at-base predicate. If the robot is not at a known location a new location is created. Based on Speck et al.'s work [11].
blocked_locations	When a robot's laser scanner senses an obstacle within the robot's planned path, the blocked location is added to the problem to allow the robot to drive to the obstacle and use its RGB camera to inspect it. Blocked locations are removed from the state when they are no longer required. (Example shown in Listing 2)
sensed_obstacles	This state creator adds PDDL statements of any objects on the path with identified nature. Identification can come through object recognition algorithms on the RGB camera, or directly from the cloud (e.g. closed door). Example PDDL output shown in Listing 3. Objects that have been acted-on are removed from the problem.

TABLE II

DESCRIPTION OF THE DIFFERENT ACTION EXECUTORS.

Action executor	Example action	Description
drive_base	drive-base robot1 waypoint1_room0 doorway3_room0	Retrieves the position (e.g. for doorway3_room0) from numeric fluents in the problem and commands the robot to move to that position. Based on Speck et al.'s work [11].
inspect_object	inspect-object robot1 blocked_loc1 doorway1_room2	Starts the object detection node, if it is not already running, and rotates the robot until it is facing the object. In the example, robot1 will inspect the object at blocked_loc1.
push_box	push-box robot1 blocked_loc1 doorway1_room2 box1	The robot (robot1) will push the box (box1) to a position where it no longer blocks the robot from getting to the target location (doorway1_room2).
remote	open-door remote doorway0_room0 doorway0_room1 door0	Any planned action whose first parameter is not equivalent to the local robot, is executed by this action executor.

inspect-object action is picked up by the sensed_obstacles state creator.

The planner will now compare the updated state with the preconditions of the next planned action, namely the drive-base action. Because the updated state violates the preconditions (the can-move-to rule), replanning is triggered. The domain enricher will check the updated problem of Listing 3 and notice that there is an object of type box. It will copy any actions on this type of object into the PDDL domain, in our exemplary scenario a box only has the push-box action.

```
(:objects  box1 - box )
(:init
 (is-blocked-by box)
 (object-is-in-path box1 blocked_loc1
    ↪doorway2_room3)
 (= (x box1) 12.4)
 (= (y box1) 1.0)
 (= (width box1) 1)
 (= (height box1) 1)
)
```

Listing 3. PDDL problem showing box1 is blocking the robot from navigating between blocked_loc1 and doorway0_room1.

The new plan is shown in Fig. 4-C: the robot will push the box out of the way and reach its goal.

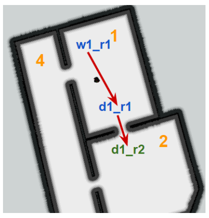
2) Smart environment sensing and actuation: In this scenario, the robot is tasked to move from room 1 to room 2. Initially, the robot only knows the static map (walls and waypoints) and thus generates a very simple plan containing two drive-base actions, see Figure 5-A. This plan is submitted by the Context Monitoring node to a plan validator in the cloud. The cloud reasons on the path in the plan and starts interpreting data of relevant sensors along the path. At one moment, the sensor for the door between room 1 and room 2 detects the door has been closed. As this state change invalidates the current plan, the plan validator will send this information to the IoT Listener node, that in turn publishes this on a topic which the Context Monitoring node is subscribed to. The Context Monitoring node pre-empts the drive-base action, although the robot by itself reports no failures.

```
(:durative-action open-door
 :parameters (?r - robot ?s - location ?g -
    ↪location ?d - door)
 :duration (= ?duration 1000)
 :condition (and
   (at start(object-is-in-path ?d ?s ?g))
   (over all (not (is-local ?r)))
   (over all (is-actionable ?d))
 )
 :effect (and
  (at end(not(object-is-in-path ?d ?s ?g)))
 )
)
```

Listing 4. open-door action from the door object's PDDL domain.

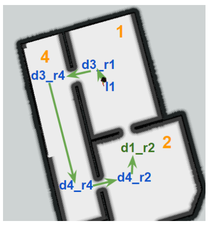
Given that the smart environment has already identified the blocking obstacle as a door, there is no need to first inspect the object. Just as in the previous scenario, the sensed_obstacles state estimator adds a door object to the PDDL problem (very similar to Listing 3). The cloud also provides the sensed_obstacles state estimator with knowledge about if the door has an electronic opener. This is set in the problem using the is-actionable predicate, which is a precondition of the open-door action. As replanning is required, the domain enricher will immediately load the open-door action definition, see Listing 4 and Figure 5-B.

(A)



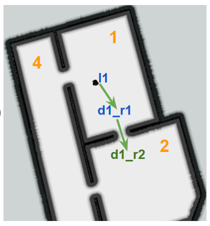
1. (drive-base r1 w1_r1 d1_r1) ← action pre-empted causing replanning
2. (drive-base r1 d1_r1 d1_r2)

(B1)



1. (drive-base r1 l1 d3_r1)
2. (drive-base r1 d3_r1 d3_r4)
3. (drive-base r1 d3_r4 d4_r4)
4. (drive-base r1 d4_r4 d4_r2)
5. (drive-base r1 d4_r2 d1_r2)

(B2)



1. (drive-base r1 l1 d1_r1)
2. (open-door remote d1_r1 d1_r2 door1)
3. (drive-base r1 d1_r1 d1_r2)

Fig. 5. Rviz displaying the robot's costmap, location and goal, alongside the task plans used when an obstacle (door1) is detected by an IoT sensor. The robot's initial location is waypoint1_room1 and has been assigned the goal: (at-base doorway1_room2 robot1). B1 shows the plan when no devices are able to open door1; and in B2 a remote IoT actuator can open door1.

The robot itself does not have the ability to open doors (i.e. (over all (not (is-local ?r))) precondition), therefore must use an alternative route or ask a remote IoT actuator to open the door between room 1 and room 2. This depends on if this door is actuatable by the smart environment.

If the door cannot be opened, the planner decides to use an alternative route via room 4 (Fig 5-B1). As there are no obstacles along this route the robot is able to reach the goal location without further replanning. If no alternative route exists, the planner will fail to find a task plan.

If the door can be opened remotely, the resulting plan contains an open-door action, rather than take the longer alternative route. This plan is shown in Fig. 5-B2. The open door request is sent through the remote action executor. When this action has completed, the sensed_obstacles state creator removes door1 from the planner's state in order to keep the PDDL problem file as compact as possible.

V. Conclusion and Future work

We have presented a generic system which improves robots' ability to plan its operations in dynamically changing environments. Observations made by sensors in smart environments allow robots to pre-empt their plan when these changes occur. With the aid of IoT actuators a robot is able to complete tasks it would otherwise be incapable of performing. Simulated experiments show this works for both IoT sensed and robot sensed obstacles. Planning time is kept short by reducing the amount of knowledge required upfront in the PDDL domain and problem. When obstacles in a robot's path are detected, a robot is able to expand this knowledge.

Further research into more intelligent upfront knowledge provisioning for problem generation could reduce the number of times a robot needs to replan. This replanning could be speed-up by exploiting knowledge from previous planning iterations. We will also study more advanced plan validation techniques, e.g. by the use of ontologies to be able to generically determine which smart environment sensors may provide useful information for the current plans. We will also study capability reasoning to determine which devices can perform an action and introduce the notion of costs. For example, there could be two robots in the neighbourhood that have the hardware capabilities to open a door, but they might be unavailable or far away.

Acknowledgements

Helen Harman is an SB fellow at FWO (project number: 1S40217N). Part of this research was funded via imec's ACTHINGS High Impact Initiative.

References

[1] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[2] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, "Semantic attachments for domain-independent planning systems," in *19th Intl Conf on Automated Planning and Scheduling*, 2009.

[3] R. P. Petrick and A. Gaschler, "Extending knowledge-level contingent planning for robot task planning," in *Workshop on Planning and Robotics (PlanRob) at the Intl Conf on Automated Planning and Scheduling*, pp. 157–165, 2014.

[4] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtós, and M. Carreras, "Rosplan: Planning in the robot operating system.," in *ICAPS*, pp. 333–341, 2015.

[5] C. Dornhege and A. Hertle, "Integrated symbolic planning in the tidyup-robot project.," in *AAAI Spring Symposium: Designing Intelligent Robots*, 2013.

[6] L. Riazuelo, M. Tenorth, D. Di Marco, M. Salas, D. Gálvez-López, L. Mösenlechner, L. Kunze, M. Beetz, J. D. Tardós, L. Montano, *et al.*, "Roboearth semantic mapping: A cloud enabled knowledge-based approach," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 432–443, 2015.

[7] M. Broxvall, M. Gritti, A. Saffiotti, B.-S. Seo, and Y.-J. Cho, "Peis ecology: Integrating robots into smart environments," in *Robotics and Automation, 2006. ICRA 2006.*, pp. 212–218, IEEE, 2006.

[8] A. Hornung, S. Böttcher, J. Schlagenhauf, C. Dornhege, A. Hertle, and M. Bennewitz, "Mobile manipulation in cluttered environments with humanoids: Integrated perception, task planning, and action execution," in *14th IEEE-RAS Intl Conf on Humanoid Robots (Humanoids)*, pp. 773–778, IEEE, 2014.

[9] J. Buehler and M. Pagnucco, "Planning and execution of robot tasks based on a platform-independent model of robot capabilities," in *Proceedings of the 21st European Conf on Artificial Intelligence*, pp. 171–176, IOS Press, 2014.

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, 2009.

[11] D. Speck, C. Dornhege, and W. Burgard, "Shakey 2016 - how much does it take to redo shakey the robot?," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 1203–1209, 2017.