Integrating Planning, Execution and Monitoring for a Domestic Service Robot

Elizaveta Shpieva, Iman Awaad

Department of Computer Science
Bonn-Rhein Sieg University of Applied Sciences
Grantham-Allee 20
Sankt Augustin 53757
Germany
elizaveta.shpieva@smail.inf.h-brs.de
iman.awaad@h-brs.de

Abstract: Embodied artificial agents operating in dynamic, real-world environments need architectures that support the special requirements that exist for them. Architectures are not always designed from scratch and the system then implemented all at once, but rather, a step-wise integration of components is often made to increase functionality. Our work aims to increase flexibility and robustness by integrating a task planner into an existing architecture and coupling the planning process with the pre-existing execution and the basic monitoring processes. This involved the conversion of monolithic SMACH scenario scripts (state-machine execution scripts) into modular states that can be called dynamically based on the plan that was generated by the planning process. The procedural knowledge encoded in such state machines was used to model the planning domain for two RoboCup@Home scenarios on a Care-O-Bot 3 robot [GRH+08]. This was done for the JSHOP2 [IN03] hierarchical task network (HTN) planner. A component which iterates through a generated plan and calls the appropriate SMACH states [Fie11] was implemented, thus enabling the scenarios. Crucially, individual monitoring actions which enable the robot to monitor the execution of the actions were designed and included, thus providing additional robustness.

1 Introduction

The RoboCup@Home competition is an annual robot competition aimed at spurring innovation in service robotics. It is composed of a series of tests (or scenarios) for specific capabilities that gradually lead to tasks requiring increasing levels of integrated functionality. As the robot is expected to function in a dynamic environment, and to accomplish objectives which may not necessarily be known before hand, it is vital to have a system that can generate plans and robustly execute them. In a dynamic environment this means tasks' execution must be monitored in order to identify when it doesn't go according to plan and thus enable the handling of such situations.

Our work was carried out on Jenny, a Care-O-Bot 3 [GRH+08] and a member of the b-it-bots RoboCup@Home team [DFH+13]. In the pre-existing architecture, the sequence

of actions which were necessary to accomplish the tasks for each scenario were hand-coded into monolithic SMACH state machines (one per scenario). SMACH is a Python library that is used to build hierarchical state machines that enable complex robot behavior [Fie11]. Such a system meant that there was limited flexibility and little possibility of re-use. Robustness in handling the dynamic environment was also limited. Since the means to handle problems at execution time were also limited, little existed in terms of task monitoring (no re-planning was possible and failures while grasping objects, for example, were handled simply by retrying, regardless of the cause of the failure).The scripts are difficult to maintain since any changes in the scenario would necessitate changes be made to the state machines and the states within them by hand.

In this paper, we present our experience in enabling Jenny to plan and robustly execute the plans (through a monitoring system) thus providing her with the flexibility one would expect of a domestic service robot. Section 2 reviews how decisions are made by other domestic service robot systems and a number of execution monitoring approaches. Section 3 motivates the integration of a planner into the architecture and the choice of the JSHOP2 planner in particular. It also explains the process by which the planning domain was created. Section 4 details the transformation of the execution process and the extension of the monitoring capabilities of the robot. Finally, experimental results are presented in Section 5 before we conclude the paper and lay out our future work in Section 6.

2 Related Work

The RoboCup@Home league continues to attract an increasing number of teams. These teams strive to improve both the robots' skills and robustness. The vast majority use some form of scenario-driven approach similar to our original implementation; others can usually be seen as approaches that follow certain policies (rules that guide decisions on actions given the current situation). Since they do not include a task planner, they share similar disadvantages such as the lack of both flexibility and robustness in dealing with unknown scenarios and dynamic environments.

The *NimbRo* team uses a four-layer architecture, built on top of ROS [QCG+09], with a task layer at the top followed by a subtask layer which uses hierarchical finite state machines, an action-perception layer and a sensori-motor layer at the bottom [DGH+12]. The *Homer* team uses task modules which correspond to simple tasks which are triggered by speech [SKS+13]. The *Golem* team robot uses their interaction-oriented cognitive architecture [Pin13]. The central module of the architecture is the Dialogue Model. The task is described by a set of situations (themselves defined in terms of expectations which can be dynamically determined). The agent decides which action is to be taken based on the expectations. These are augmented with reactive modules to handle the dynamic nature of the environment. *ToBI* uses a domain-specific library which links perception and action. Strategies are generated from robot skills/actions. These strategies are given to the decision-making system. Currently, they focus on the reusability of strategies using a coordination engine which acts as a sequencer [ZWS13]. The *WrightEagle* [CWS+13] team's system is based on Answer Set Programming (ASP). For the simple scenarios with few

steps, this system functions well, however, when the number of steps in a plan increases, the ASP solver takes an unreasonable amount of time to generate a plan. For this reason, they combine HTN planning with their ASP method.

Tech United also formulates a goal and uses a reasoner to query the initial state in order to generate a plan through a basic search process for some of the scenarios (such as the general purpose robot challenge). The decisions on which actions to perform to achieve a goal are still for the large part implemented as SMACH state-machines [LCDE13] as in our original approach.

The most closely related work is probably that of the *eR@sers* team [OOWS12] which has integrated a task planner into their system and a monitoring process. The main contribution of the work presented in [OKAM13] is the extension of the PDDL planning domain with monitoring information. A component was then implemented to convert the extended-PDDL domain to SMACH. In order to execute the same PDDL plan on another robot, this component must be adjusted to use the new robot's API. Since we keep the planning and monitoring systems separate, our approach enables us to use any off-the-shelf planner to generate the plan. This plan can then be sent to any robot with the same capabilities by calling the robot-specific SMACH states.

In addition to the decision-making process above, we are also interested in the monitoring process to increase robustness. As the system is built on ROS, we benefit from the various forms of monitoring it provides. These can generally be categorized as node-based monitoring (through the node-monitoring stack [Nie11]) and hardware-level monitoring (through the diagnostics stack). The robot monitor [Sai13] displays diagnostic messages about the state of hardware (for instance, if a wheel breaks or a camera is not plugged in). While these components are valuable, they are not capable of monitoring the specific action execution of a robot.

Task execution monitoring components traditionally use approaches that compare a predefined model of the world after the action is executed with the observed state of the world [BKS08]. While some approaches monitor the effects of planning operators, a variety of other states are monitored in other approaches. [PM99] uses rational-based monitoring where features of the environment that can influence the plan before execution are identified and their state is monitored. In [FSW] plan invariants (conditions that hold during the execution of the plan) are monitored. Using these invariants invalid plans can be detected early and the agent is able to react to changes in the environment. The LAAS [LI04] and ROGUE [HV98] architectures for example, are behavior-based approaches where the monitoring actions are modeled beforehand, as in our approach.

3 The Planning Process

Automated planning is the process by which artificial agents generate sequences of actions (plans) to reach their goals [GN04]. To do this, planners need a domain description which includes the actions that an agent can take, when they may be taken and how the execution of the actions change the world.

Real-world domains, like the domestic service domain, are difficult to model in classical planning representations. The HTN planning approach uses the inherent hierarchy which exists in many domains to decompose the planning problem. It recursively decomposes **non-primitive tasks** using recipes called **methods** until **primitive tasks** are reached. **Operators** then directly perform these primitive tasks. It enables us to encode expert knowledge for accomplishing a task (i.e. the best way to perform it) and therefore provides plans that perform tasks in the 'best way'. For instance, if more then one object needs to be moved from one location to another the modeler can provide 'the expert knowledge' that the objects should be placed together on a tray and then delivered. To fetch an object for a user, one would intuitively decompose the task into fetching the object and then delivering it to the user. This intuitive way of modeling is one of the main advantages of the approach. Moreover, HTNs enable human users to understand why a given action is being performed (the link between a subtask and the actions is clear). This is especially important as humans and agents co-inhabit the environment in the domestic service domain.

In our work, we use the JSHOP2 implementation [Ilg06] of the HTN algorithm [EHN94]. The main reason for preferring JSHOP over SHOP [NCLMA99] is that it generates plans quicker, an important requirement in real-world, dynamic environments. A detailed comparison of the two planners can be found in [IN03].

The first step in integrating the planner was to model the domain. This remains a complex and time-consuming task, but is made simpler as the procedural knowledge that is encoded in the monolithic SMACH state machines could be re-used to create the HTN planning methods and operators. Flexibility and re-use are achieved by decomposing the monolithic scenarios into modular parts.

Let us take the 'bring object' scenario as a use case. In this scenario, the robot is asked to fetch a given item and return it to the user. The initial SMACH state machine for this scenario is shown in Figure 1. To perform this task, the robot requires knowledge of its position, the position of the given object within the environment and the position to which it needs to return the object (in this case, the person). The goal statement which is provided to the planner instructing it that it should fetch the Pringles ™and return them to a person known as 'guest' is given as: `(bring_object Jenny pringles guest)`, where `bring_object` is the name of our method **method**, `Jenny` is the value of first parameter: *?robot*, `pringles` is the value of second parameter *?object* and `guest` is the value of third parameter *?to* and refers to the known person to whom the Pringles ™should be delivered.

From the SMACH state for this scenario, 13 methods and 7 operators were created in the JSHOP2 syntax (a part of the resulting task network generated by the planner can be seen in Figure 2). The task was decomposed into two subtasks: *fetch an object* and *deliver an object* where the robot first gets the object and then delivers it to the person. The first method `bring_object` decomposes the *fetch object* subtask and has only two main parameters: what to deliver and to whom it should deliver it. The information about the desired object is easily obtained from the initial state. The modularity introduced by refactoring the scenario means that the same methods and operators can be re-used in order to carry out other tasks. The third parameter *?robot* was added to create a robot-independent plan which might be used on another mobile manipulator with similar capabilities.

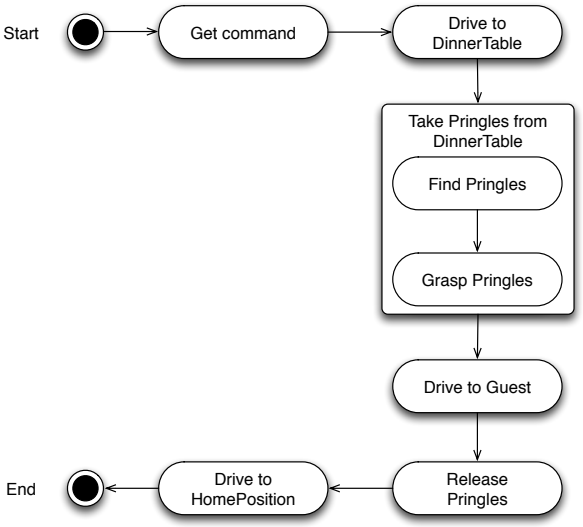

Figure 1: The monolithic state machine for the 'bring an object' scenario.

4 The Execution and Monitoring Processes

Having generated a plan, it now needs to be executed. For each JSHOP2 operator, a corresponding simple SMACH state machine was created. These SMACH state machines were then wrapped in the ROS *actionlib* [EME13] which enables them to be called as needed. A *task management* component was created that iterates through the generated plan and calls the appropriate SMACH state which in turn executes the action (through the *execution* module). After calling one of the plan action's SMACH states, it waits to receive confirmation of that action's execution status before proceeding to call the following action.

Figure 3 presents an overview of the software architecture of the system. The process starts with the user giving the robot a task to accomplish (or alternatively, a component which decides which goal to achieve). The speech recognition process parses the command and the *scenario manager* converts this command to a goal statement for the planner. As the planner is not yet integrated into a ROS node, and the knowledge base (KB) is still being developed, the initial state has been manually created and together with the goal statement has been input to the planner. Plans have been generated offline and stored in different files. The *scenario manager* currently decides which of the two pre-generated plans (corresponding to the two scenarios which have been refactored) is the one being specified by the given command. The task is sent to the *task dispatcher* within the *task management* module (specifying the correct file with the corresponding plan). Once the planner is integrated within ROS, and the KB completed, the goal along with the initial state (queried from the knowledge base) would be used as input to the planner and the

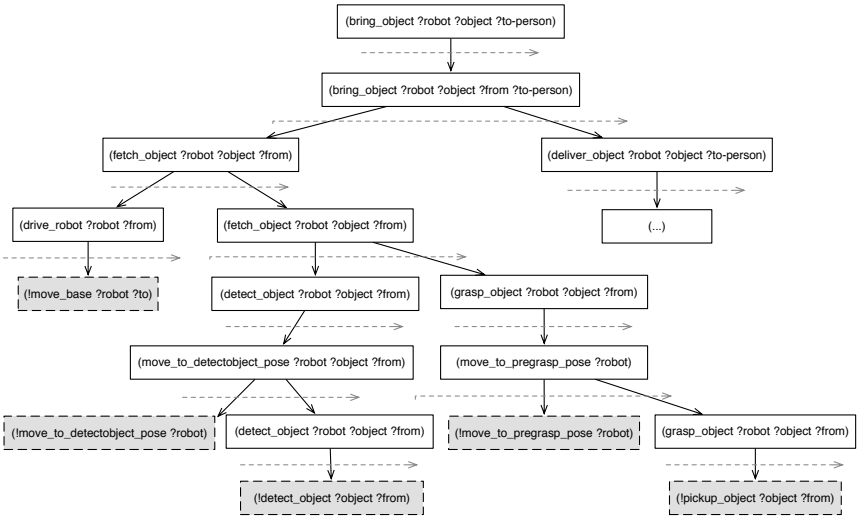

Figure 2: The task network for the 'bring object' task (structure based on that of [GN04])

resulting plan would be passed directly to the task dispatcher as described above.

The *task management* module contains the `task dispatcher` module which takes an action from the plan and sends it to the `task executor` module. The `task executor` parses the action into 'name' and 'parameters'. These are then sent to the *execution* module as input parameters. The corresponding action is called by the action server wrapper and the parameters are sent as 'keys'. Each action within a plan has a SMACH state machine with three corresponding SMACH states: the first executes the action, the second executes the monitoring action and the third reports the outcome of the SMACH state machines a whole (a concrete example of the `move_base` state machine can be found in Section 4.2).

For the action server wrapper `as_move_base` (Listing 1), the *execution* module has two outputs: 'feedback', which provides the up-to-date state of an action's execution, and 'result', which provides the final status after the monitoring action has been executed. These are sent by the `executor` back to the `task dispatcher` module which then decides if the next action should be sent for execution (Section 4.1 details the possible feedback and result values) . Wrapped within each action server wrapper is an action state which directly communicates with the robot's software components.

```
as_move_base = ActionServerWrapper('move_base',
    brsu_task_manager.msg.TaskManagerAction, move_base,
    succeeded_outcomes = ['pose_reached'],
    aborted_outcomes = ['overall_failed_reach_pose'],
    feedback_slots_map = {'server_result' :'action_feedback'},
    feedback_key = 'action_feedback',
    goal_key = 'pose',result_key = 'result_move_base')
```

Listing 1: The action server wrapper for the move base action stating the possible outcomes and the

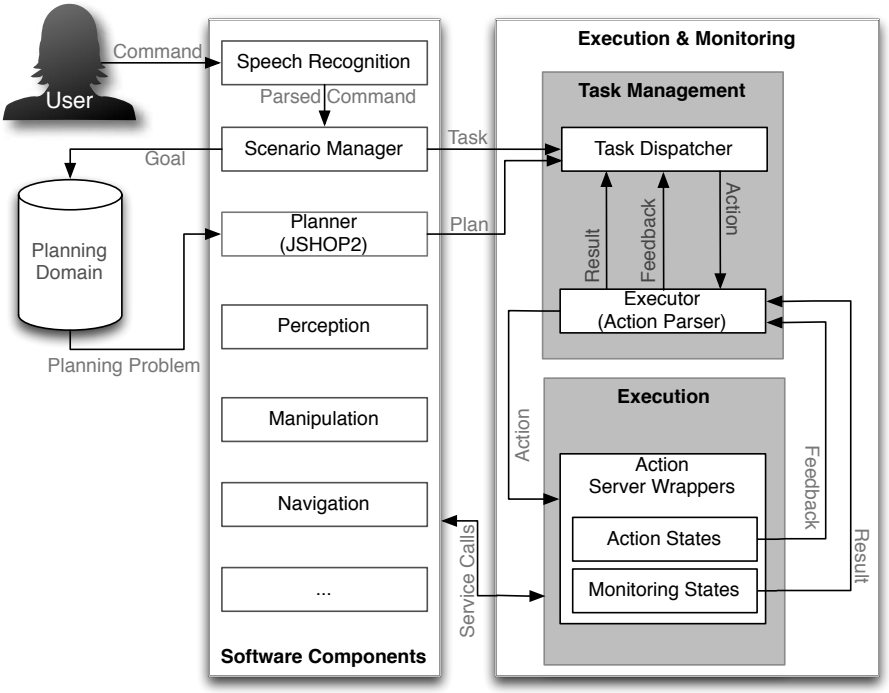

Figure 3: An overview of the planning, execution and monitoring systems

various parameters. $goal\_key = 'pose'$ can be any known position on the map.

4.1 Action-Execution Outcomes

To each new modular SMACH action state, action-monitoring functionality was added. This ensures that the status of an action's execution is always known and can be propagated throughout the system to enable more intelligent handling of faults. In addition, it signals to the `task dispatcher` when the next action may be sent for execution. To re-iterate, feedback is sent following the execution of the action, while results are sent following the execution of the monitoring action(s). The state machine's status and the transitions between states can be visualized using the SMACH Viewer [Boh11] tool.

Let us consider the ideal situation when executing actions. The *execution* module receives the two inputs: the action name and the parameters. The action server wrapper takes them and calls the corresponding action. The robot tries to perform the action and at some point it believes that it has finished the execution of the action successfully and returns the 'feedback' that it has **finished**.

This may not always be true however and to ensure that the action has indeed succeeded we must check this by executing a monitoring action. Once the appropriate monitoring action (or possibly actions) is completed, we may be more certain that the robot has indeed performed the task successfully and provide the 'result' that the action is **verified**.

Executed plans do not always succeed. If an action fails to be executed for whatever reason, 'feedback' that the action has **failed** is sent (also in the case that monitoring action itself fails).

The chances of errors and failures occurring during execution in the real-world are significant. These may be due to some sensor noise, calibration error or even due to the actions of the user after the robot has finished executing an action (for example, by moving an object that was about to be grasped). It could very well be that the action was executed (with the 'feedback' sent as **finished**) , and the monitoring action then executed, only to find that the monitored effects of the action do not match its expectation. The 'result' of the action is then **aborted**.

Currently, in case of an action being **failed** or **aborted**, the robot maintains its current pose until it receives further instructions form the user. Determining what a safe policy may be is a vital research area and still an open issue. In the future, one can imagine that in most situations, a new plan may safely be generated and sent for execution, perhaps after consultation with a user.

In some cases, a number of retries may be specified for actions which are prone to failure, as in the `pick up object` task. The robot has the opportunity to retry an action which could not be verified. This could happen when the robot attempts to grasp an object and the object slips out of its hand for example. All effectors would be in the desired pose, but the object would not be within the gripper. The action will be retried until the monitoring action returns **verified** or after a user-defined number of trials.

Figure 4 provides a summary of the possible outcomes and expected behavior. The semantics of these outcomes are summarized here:

- **Finished**: The action was sent for execution and accepted by the relevant component. The action's success or failure has not yet been monitored.
- **Verified** : The action is verified through the monitoring actions. Action execution should now be considered successful.
- **Failed**: The action has failed due to an internal error. The robot maintains its pose until further instruction from the user.
- **Aborted**: The action is aborted because the monitoring actions do not detect the success of the executed action.
- **Retrying**: The action will be retried.

A concrete example of an action and its monitoring action which demonstrates the outcomes presented above is given in the following subsection.

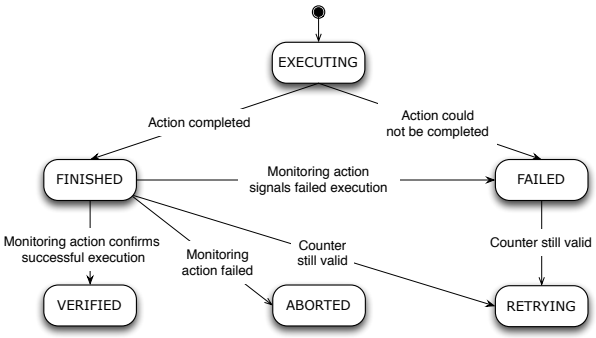

Figure 4: An overview of the possible outcomes for actions and monitoring actions.

4.2 The Move Base Example

As we mentioned in Section 4, each planning operator has a corresponding SMACH state machine which consists of three SMACH states.

Listing 2 shows the SMACH state machine for the `!move_base` operator. If the action has **finished** the monitoring action will be performed. The `move_base` state has $'pose'$ as input parameter and $'action\_feedback'$ as feedback data to the *task management* module. If the action **failed** then the `status_report` state is executed. This state sends the messages on the outcome of the action's execution. The `move_base_monitor` state has three possible transitions. If the action's execution is considered successful, i.e the robot has reached its desired pose, then the action is **verified**. The transition 'overall_failed_reach_pose' would mean that the action was **aborted** and the robot then maintains its current position as explained above. The remapped 'result_move_base' data is the feedback that was sent to the *execution* module. A monitoring action or set of actions is similarly implemented for each operator, depending on its functionality, and the same protocol is followed.

```
smach.StateMachine.add('move_base', move_base(),
  transitions={'finished':'move_base_monitor',
       'failed':'status_report'},
  remapping={'message':'message','pose_in':'pose',
       'state_results':'action_feedback',
       'move_base_state':'move_base_state'})

smach.StateMachine.add('move_base_monitor', move_base_monitor(),
  transitions={'verified':'pose_reached',
       'aborted':'overall_failed_reach_pose',
       'failed':'status_report'},
  remapping={'state_results':'result_move_base',
       'move_base_state':'move_base_state'})
```

```
smach.StateMachine.add('status_report', status_report(),
   transitions={'failed':'overall_failed_reach_pose'},
   remapping={'state_results':'result_move_base',
        'move_base_state':'move_base_state'})
```

Listing 2: The move_base state machine which consists of the action, action monitoring and status report SMACH states.

5 Results

This work aims to validate the integration of a task planner into a pre-existing system. This was done to enable the agent to plan and monitor the actions' execution thus, increasing flexibility and robustness. The integration necessitated the refactoring of the existing execution process, and the creation of both the action monitoring system and the planning domain. The modularity of these implementations make them highly re-usable.

The experiments were aimed at testing whether the system presented here produced the same agent behavior as the pre-existing system did. Experiments were performed in the Gazebo simulator [KHD11] for the two scenarios: bring an object and guide a robot. Figure 5 shows a snapshot of one of the experiments where Jenny should bring an object. The outcomes of the actions' execution can be seen in the window in the bottom left corner, including the unsuccessful execution of the action to grasp an object and the information provided by the SMACH monitoring state that the action is being retried.

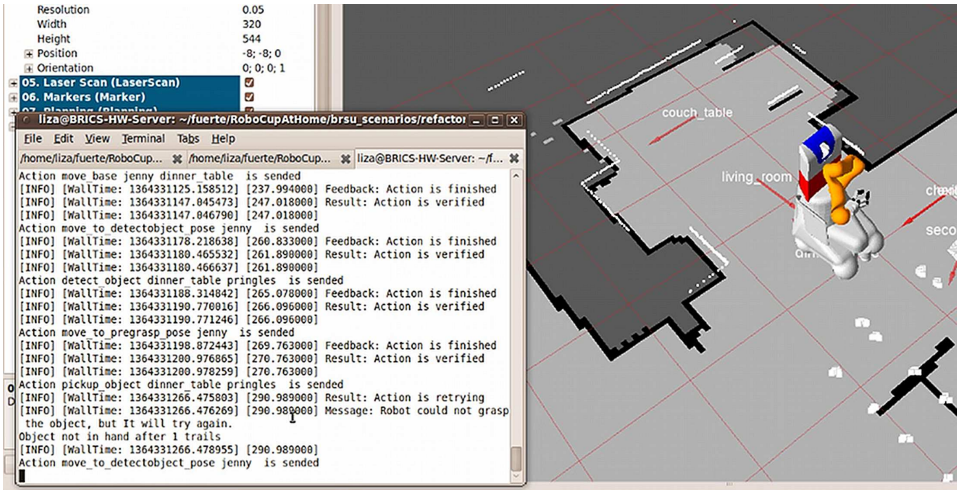

Figure 5: A screenshot of Jenny performing the bring an object task.

The planner successfully generated a plan for each of the tasks. The actions in the plans were read, executed, monitored and reported. The system performed as the original did in

terms of robot behavior, with the added benefit of the monitoring process which performed as described in this work. The time to execute each of the scenarios in simulation using the original and current implementation was the same (10 minutes for the 'bring an object' scenario and 6 minutes for the 'guide a robot' scenario).

Table 1 shows the number of solution plans generated by JSHOP2 for each scenario, the time it took to generate them, and the length of the shortest plan. While three plans were generated for the first scenario, they were identical. The reason they were considered to be three plans by JSHOP2 is because the task networks which led to the plans were different. The complexity of generating a plan is at its minimum when there is exactly one way to decompose a task. In this case, for example, the task could be decomposed using the `grasp_object` method with three parameters `(?robot ?object ?from)`, or the `grasp_object` method with two parameters `(?object ?from)`, leading to different task networks which ultimately however resulted in the same plan.

	Bring an Object	Guide robot
Number of solution plans	3	1
Time to plan	3 seconds	1.5 seconds
Minimum plan length	8	5

Table 1: The statistics on the plan generation process for the two scenarios presented here.

For an idea of the size of the software packages, the source lines of code (SLOC) metric is commonly used. The task management and execution components have 93 SLOC and 1947 SLOC respectively. The execution component's SLOC size is larger as it contains the description of all robot actions. The original implementations of the two scenarios had 2800 SLOC.

6 Conclusions and Future Work

Flexibility, reusability, extensibility, and robustness are qualities that we strive for in software systems. We have increased these qualities in our system by first integrating a planner, second by refactoring the execution process to create a modular, re-usable and extensible planning domain and its accompanying simple SMACH state machines, and finally by taking first steps towards an execution monitoring system.

With the proof of concept obtained here, we will extend the planning domain for our service robot. The extraction of the expert knowledge on how tasks should be accomplished from the pre-existing scenario state machines, and the use of this knowledge to build the HTN planning domain was extremely helpful. This top down approach which started by analyzing the scenarios to identify tasks and then actions will be accompanied by a bottom-up approach. We will start by analyzing the basic actions that a robot can perform. These cations can be simple, such as moving a specific actuator, to more complex ones such as pouring or spraying. This will facilitate the future design of more general, non-primitive

tasks using these capabilities (such as cleaning the room) by combining the actions. We will also investigate the ability to preempt/interrupt the execution process in a safe way.

Currently, the JSHOP2 planner remains outside of the system. A ROS node for the task planner will be created. The same needs to be done for the KB. A more holistic monitoring, fault detection and diagnosis system would then serve to ensure a much higher level of robustness.

7 Acknowledgements

The author I. Awaad would like to acknowledge the financial support provided by a PhD project scholarship of the Department of Computer Science of Bonn-Rhein-Sieg University. The authors thank the reviewers, Nico Hochgeschwender and Sven Schneider for their helpful feedback.

References

[BKS08] Abdelbaki Bouguerra, Lars Karlsson, and Alessandro Saffiotti. Monitoring the execution of robot plans using semantic knowledge. *Robotics and autonomous systems*, 56(June 2008):942–954, 2008.

[Boh11] Jonathan Bohren. SMACH Viewer Documentation. Online at http://www.ros.org/wiki/smach_viewer, July 2011.

[CWS+13] Xiaoping Chen, Feng Wang, Hao Sun, Jiongkun Xie, Min Cheng, and Kai Chen. KeJia: The Integrated Intelligent Robot for RoboCup@Home. 2013.

[DFH+13] R. Dwiputra, M. Fueller, F. Hegger, N. Hochgeschwender, J. Paulus, I. Awaad, S. Schneider, A. Bierbrauer, E. Shpieva, I. Ivanovska, N. V. Deshpande, A. J. Gaier, A. Hagg, J. M. Sanches Loza, A. Y. Ozhigov, P. G. Ploeger, and G. K. Kraetzschmar. The b-it-bots RoboCup@Home Team Description Paper. 2013.

[DGH+12] David Droeschel, Kathrin Gräve, Dirk Holz, Michael Schreiber, and Sven Behnke. NimbRo @Home Team Description. 2012.

[EHN94] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN Planning: Complexity and Expressivity. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1123–1128. AAAI Press, 1994.

[EME13] Vijay Pradeep Eitan Marder-Eppstein. Actionlib Documentation. Online at http://www.ros.org/wiki/actionlib, July 2013.

[Fie11] Tim Field. SMACH Documentation. Online at http://www.ros.org/wiki/smach/Documentation, July 2011.

[FSW] Gordon Fraser, Gerald Steinbauer, and Franz Wotawa. Plan Execution in Dynamic Environments. In Moonis Ali and Floriana Esposito, editors, *IEA/AIE*, pages 208–217.

[GN04] M. Ghallab and D. Nau. *Automated Planning: theory and practice*. Morgan Kaufmann Publishers, 2004.

[GRH+08] Birgit Graf, Ulrich Reiser, Martin Hägele, Kathrin Mauz, and Peter Klein. Robotic Home Assistant Care-O-bot Â 3 - Product Vision and Innovation Platform. 2008.

[HV98] KZ Haigh and MM Veloso. Planning, Execution and Learning in a Robotic Agent. *AIPS*, 1998.

[Ilg06] Okhtay Ilghami. Documentation for JSHOP2. Technical Report CS-TR-4694, University of Maryland, College Park, MD 20742 USA, May 2006.

[IN03] Okhtay Ilghami and Dana S. Nau. A General Approach to Synthesize Problem-Specific Planners. Technical Report CS-TR-4597, UMIACS-TR-2004-40, University of Maryland, October 2003.

[KHD11] Nate Koenig, John Hsu, and Mihai Dolha. Gazebo Documentation. Online at http://gazebosim.org/documentation, July 2011.

[LCDE13] J. J. M. Lunenburg, S. A. M. Coenen, S. Van Den Dries, and J. Elfring. Tech United Eindhoven Team Description. 2013.

[LI04] Solange Lemai and F Ingrand. Interleaving temporal planning and execution in robotics domains. *AAAI*, pages 617–622, 2004.

[NCLMA99] Dana S. Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 968–975, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[Nie11] Tim Niemueller. ROS Nodes Monitoring Documentation. Online at http://www.ros.org/wiki/node_monitoring, July 2011.

[OKAM13] Kei Okada, Yohei Kakiuchi, Haseru Azuma, and Hiroyuki Mikita. Task Compiler: Transferring High-level Task Description to Behavior State Machine with Failure Recovery Mechanism. In *Workshop on Combining Task and Motion Planning at the IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[OOWS12] Hiroyuki Okada, Takashi Omori, Norifumi Watanabe, and Takayuki Shimotomai. Team eR@sers in the @Home League Team Description Paper. 2012.

[Pin13] Luis Pineda. The Golem Team, RoboCup@Home. 2013.

[PM99] Martha E. Pollack and Colleen McCarthy. Towards Focused Plan Monitoring: A Technique and an Application to Mobile Robots, 1999.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully B. Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[Sai13] Isaac Saito. ROS Robot Monitor Documentation. Online at http://ros.org/wiki/robot_monitor, July 2013.

[SKS+13] Viktor Seib, Florian Kathe, Daniel Mc Stay, Stephan Manthe, Arne Peters, J Benedikt, Raphael Memmesheimer, Tatjana Jakowlewa, Caroline Vieweg, Sebastian St, M Simon, Alruna Veith, Michael Kusenbach, Malte Knauf, and Dietrich Paulus. RoboCup 2013 - Homer @UniKoblenz. 2013.

[ZWS13] Leon Ziegler, Jens Wittrowski, and M Schöpfer. ToBI-Team of Bielefeld: The Human-Robot Interaction System for RoboCup@Home. 2013.