Task Planning for an Autonomous Service Robot

Thomas Keller, Patrick Eyerich, and Bernhard Nebel

University of Freiburg
Institut für Informatik
Georges-Köhler-Allee 52
79111 Freiburg
{tkeller,eyerich,nebel}@informatik.uni-freiburg.de

Abstract. In the DESIRE project an autonomous robot capable of performing service tasks in a typical kitchen environment has been developed. The overall system consists of various loosely coupled subcomponents providing particular features like manipulating objects or recognizing and interacting with humans. To bring all these subcomponents together to act as monolithic system, a high-performance planning system has been implemented. In this paper, we present this system's basic architecture and some advanced extensions necessary to cope with the various challenges arising in dynamic and uncertain environments like those a real world service robot is usually faced with.

1 Introduction

The overall aim of the DESIRE[1] project [1] was to develop an autonomous robot capable of performing service tasks in a typical kitchen environment. From the project's beginnings, rather than focus on the accomplishment of predefined scenarios, it was decided to keep the system as unbounded as possible in order to gain maximum flexibility. This led to a system architecture consisting of several loosely coupled subsystems which are able to fulfill basic tasks like manipulating objects, driving autonomously in a dynamic environment or recognizing and interacting with humans. From our point of view, the most important implication of this decision was the need for a *domain-independent* planning system that is able to combine any number of these subsystems in an efficient yet stable manner.

The integration of such a task planner into the system increases the robot's level of intelligence and flexibility by altering the way the robot is controlled, moving from predefined sequences of detailed user instructions to a more sophisticated goal oriented approach. It is not longer required to provide the robot with a fully worked out description of its task (e.g., "Go to the big table, then take the plate, then return and give me the plate!") but rather to state some imperative command (e.g., "Give me a plate!") and leave it to the robot to find a suitable *plan* to achieve the according *goals* on its own, combining the features of the different components in a meaningful way.

[1] DESIRE is a project with several partners from industry and academia and is an abbreviation for "German Service Robotics Initiative".

Task planning itself is a thoroughly investigated subfield in artificial intelligence [2]. However, in a robotics context, it has to be dealt with certain aspects complicating its application, including imperfect information about the environment, non deterministic changes, or user interaction. The main contribution of this paper is to show how to overcome these problems and make task planning suitable for everyday use in a robotics context.

The remainder of the paper is structured as follows: In the next section, we present the basics of the used planning system, describe how the current situation of the robot and its surroundings can be described in the *Planning Domain Definition Language* (PDDL) and show how it can be adapted to a dynamically changing and partially unknown environment via continual planning. Section 3 covers how to bridge the gap between several independent planning episodes, while the subsequent section shows how problems containing subproblems of different granularity can be solved efficiently. Before we conclude, we present the whole system architecture of the planning subcomponent in Section 5. Related work is referred to throughout the running text whenever it fits.

2 Planning in Real-World Environments

Given an initial state, a set of actions, and a goal formula, classical planning is about finding sequences of actions turning the initial state into a state satisfying the goal formula. The planning framework we use in this paper is PDDL2.1 Level 3 [11] extended by object fluents as proposed by Geffner [10]. In the following, we will use the term *state variable* to refer to predicates, numeric fluents and object fluents.

The classical planning approach heavily relies on having a complete and certain description of the situation the agent is faced with. Furthermore, actions need to be fully deterministic and the only changes allowed to occur in the environment are due to actions the agent decides to execute. Obviously, these constraints are too restrictive when it comes to modeling a domain depicting the real world: There might be other agents altering the state of the world (including nature), actions might have stochastic or probabilistic effects and the current state of the world might not be fully known.

Typical approaches dealing with such domains are *contingent* and *probabilistic* planning. These approaches try to generate conditional plans and policies (mappings from states to actions), respectively. Unfortunately, both approaches are of much higher complexity [7, 8] than classical planning and usually fail to scale in even moderately complex scenarios. Furthermore, it might be impossible to model all potential outcomes of actions in dynamic environments, or concrete probabilities of outcomes are unknown.

Recently, we have proposed an alternative technique to deal with real world environments: *Continual Planning* (CP) [6], in which a continuous loop between *planning*, *plan execution* and *execution monitoring* is performed. In the planning phase, the agent is allowed to postpone the decision of how to fulfill subgoals to a later point in time when more information is available by using *assertions*

as part of the plan. Then, the first action of the plan is executed. In the third phase, a monitoring procedure checks whether the remainder of the plan is still executable and still fulfills the goal. If that is not the case or if the plan has to be refined since the next executable action is an assertion, the agent switches to the planning phase. Otherwise, the next action's execution is started and the system continues with monitoring it.

For DESIRE, we have integrated the CP approach within our temporal planning system which is briefly introduced in the next section. Since temporal planning allows for concurrent actions of variable durations, there might be more than one action to start in the execution phase and these actions might have different durations. Furthermore, situations might arise in which an action has already finished, while others are still running. Therefore, we have extended the monitoring component to consider such actions with their remaining durations.

2.1 Base Planning System: TFD

We use *Temporal Fast Downward* (TFD) [4] as the base planning system. TFD is a domain-independent progression search planner built on top of the classical planning system *Fast Downward* [3]. It extends the original system to support durative actions and numeric and object fluents.

TFD solves a planning problem in three phases: First, the PDDL planning task is translated from its binary encoding into a more concise representation using finite-domain variables. This enables the use of heuristics employing hierarchical dependencies between state variables. In the second step, efficient internal data structures for the heuristic and the search component are generated. The most important ones are domain transition graphs for each variable that encode how a state variable's value can be changed, and the causal graph that represents the hierarchical dependencies between different state variables. Finally, a best-first progression search is performed, guided by a numeric temporal variant of the context-enhanced additive heuristic.

2.2 Generation of the initial state and the goal description

The preferential way of assigning jobs to a service robot certainly is to formulate them as an imperative command. To process such a command, a robot needs to be able to recognize that the user's utterance is directed to him and to parse the utterance into an appropriate textual format. In DESIRE, this is done by the *automatic speech understanding* component (ASU). The ASU has a predefined set of grammar based frame structures used to map utterances to keywords and to classify them into categories like 'socialization' or 'instruction'. Additionally, single frames are mapped to classes like 'action' or 'object'. With the help of these keywords, the planner then extracts the information captured in an instruction utterance and transforms the command into a logical formula describing the goal. Basically, this is done by mapping the action frame of the frame structure generated by the ASU to its description in the PDDL domain file. The starting point of the generated goal formula is then the effect of this action. Afterwards,

the parameters of the effect are replaced with appropriate fillers extracted out of the utterance (again detected with the help of the frame structure). Parameters for which no filler exists become universally or existentially quantified, depending on the found keywords (e.g., "Bring the salt cup to me!" is transformed to $\exists s(\mathit{salt\_cup}(s) \wedge \mathit{loc}(s) = \mathit{user})$, while "Bring all salt cups to me!" is transformed to $\forall s(\mathit{salt\_cup}(s) \Rightarrow \mathit{loc}(s) = \mathit{user}))$.

The information about the current world state is distributed among the subsystems, e.g., the information about the positions and orientations of objects is present in the scene-analysis component while the position and internal status of the robot belongs to the self-model. Thus, the planning relevant information has to be gathered and unified in order to generate a coherent initial state. For that purpose, we have developed the *abstract world model* (AWM). The AWM is implemented as a plug-in mechanism and thus does not depend on the currently active components. Each component generates a proxy subcomponent which is responsible for passing the relevant information to the planner in a consistent way. During runtime, the components register their proxies with the planner, allowing the planner to query all relevant information. Based on the information gathered that way the planner generates a globally consistent initial state.

To express information about specific properties like color or shape shared by all instances of an object class, we have developed an ontology framework. The ontologies content is grounded during runtime and added to the initial state. Furthermore, for each object class a place where instances of that type can usually be found is stored in the ontology. If one or more objects obviously required to fulfill a task are missing (e.g., if the goal is to put away all salt cups but none is present), the goal is temporally changed to find these objects and the ontological information can be utilized to fulfill this temporally goal.

3 Global Memory

While Continual Planning serves as a much faster substitution for contingent planning, one problem that has not been addressed yet arises: Typically, planning systems are not designed to consecutively solve planning tasks that possibly depend on each other. Especially the success or failure of an action's execution, which is unknown during the planning episode might cause dependencies that need to be dealt with: While, in the case of an execution failure, the planning system will realize via monitoring that no progress was made in the active plans execution because the system's state did not change in the anticipated way, it might not be able to react on this because the state did not change *at all*. In this case, the planner will be confronted with the same planning task over and over again, leaving the whole system in an endless loop of generating a plan, failing in its execution, identifying this through monitoring and again generating the same, inexecutable plan.

Our solution to this problem is the *Global Memory* (GM), which keeps additional facts for each action that are added to the initial state *in the next planning episode after that actions execution*. These facts are divided in two sets, one that

maintains facts that are applied in the case of successful execution, and the other for execution failure. A fact consists of either atomic or universally quantified state variable assignments, including increase and decrease operators for numeric state variables. Depending on the feedback of the sequencer after an action's execution, the according facts are applied to the GM, and in each planner run the content of the GM is added to the initial state.

An example for the syntax we use to describe GM updates is given in Figure 1. The first line merely states the name of the assigned operator, in this case the grasp action with parameters $?m$ (a movable object), $?g$ (a gripper) and $?l$ (a location). The grasp action has two preconditions that are important with regard to the GM: The scene model of the environment needs to be up-to-date, and the number of times it was unsuccessfully tried to grasp an object $?m$ with gripper $?g$ from location $?l$ in the past may not exceed a certain number of trials (in our case, a maximum number of two trials appeared to be reasonable).

```
GRASP ?m ?g ?l
  SUCC: (not (scene_model_updated))
        (forall ?M ?G ?L ((grasp_unsuccessful ?M ?G ?L) = 0))
  FAIL: (not (scene_model_updated))
        ((grasp_unsuccessful ?m ?g ?l) += 1)
```

Fig. 1. Global Memory description of the grasp action.

Both pieces of information cannot be gathered by querying the AWM proxies of the according subarchitectures, as neither scene-model nor manipulation keep track of their history. Due to this reason, the GM is used to pass this information between consecutive planning episodes: Independent of the success of a grasp action's execution, the scene model is considered to be out-of-date, as can be seen in line 2 and 4 – in both cases, `(not (scene_model_updated))` is added to the initial state. The tracking of unsuccessful grasps on the other hand depends on the success of an action's execution: If the grasp action is executed successfully line 3 is applied and all unsuccessful grasps that ever happened are forgotten, i. e., reset to zero, since by successfully grasping any object the environment changes enough to justify retrying formerly unsuccessful grasps. In case that action's execution fails, the variable that keeps track of unsuccessful grasps is increased by one, as is stated in line 5. This increase allows us to break the endless loop the system possibly enters in case of execution failure: After a maximum number of trials to grasp an object, the according state variable `(grasp_unsuccessful ?m ?g ?l)` is bigger than the maximum number of trials given in the grasp action's precondition and the operator is no longer applicable, forcing the planner to find a plan *without* trying to grasp object $?m$ with gripper $?g$ from location $?l$ again. Of course, this kind of modeling the domain leads to situations where the robot is not able to execute its orders anymore because it already tried to grasp an object from all locations with both its grippers. For this case, we added

operators to the domain which can only be applied if some operator's execution failed more often than a given threshold allows, and which notify the user that (parts of) the received command cannot be executed.

Note that counting the number of unsuccessful action executions is obviously not the only way to break potential endless loops with the GM. Alternatively we could for instance force the planner to never use an unsuccessfully executed action again in the same state, or, in the case of grasping, the concerned docking position could be altered somewhat, but in the DESIRE scenario the described method worked sufficiently well.

4 Planning with External Modules

Planning in real-world domains requires to solve problems of different granularity: On the one hand, high-level actions like driving to a certain position or grasping a certain object are atomic actions with well-defined symbolic preconditions and effects. On the other hand, how to actually perform such an action might be a difficult subproblem in itself: To reach a certain position it is usually required to invoke a path planning subroutine, and before an object can be grasped a collision-free trajectory needs to be computed.

In previous work we have presented a new approach to solve such types of problems: The use of *semantic attachments* [5]. A semantic attachment is an external procedure called during the planning process to evaluate specific conditions or to directly alter the planning state. By using semantic attachments for subproblems like path planning we combine the advantages of both approaches while circumventing their disadvantages: on the one hand, the high-level planner does not need to care about subproblems since they are dealt with in the semantic attachments, on the other hand, only information actually needed to solve the problem is generated at the time the semantic attachment in invoked.

To deal with the mentioned issue of solving problems of different granularity, we implemented several semantic attachments, in particular for manipulating objects. When planning for *grasping* an object, it quickly falls into place that a purely symbolic representation is insufficient for the task. Having said that, the complete integration of a manipulation planner is far too inefficient, as one call to such a planner usually requires runtimes in the magnitude of seconds and in non-trivial problems hundreds to thousands of such calls are required. Therefore, we used a solution in between by utilizing an approximation procedure as a semantic attachment. This gives us more precise results than purely symbolic planning while staying efficient even in problems of considerable complexity. In dependence of the object's location and the shape of the surface it is located on, the semantic attachment checks whether a given docking position of the robot is appropriate for grasping. For that purpose, it is checked whether the object is within reach of the manipulator in question and whether it is not covered by other objects nearby it. Furthermore, it is ensured that the angle between the robot and the object's position is within some predefined range.

To find an appropriate position on a given surface to *place* an object on, we used a semantic attachment that works as follows: First, the surface is partitioned into grid cells of one square centimeter. Then, the occupied cells are determined on the basis of all other objects on the same surface. Finally, a free area big enough to hold the object and maximizing the remaining free space is chosen as the position to place the object on. Note that all these computations are performed only when they are required.

5 System Architecture

Before we conclude, we give an overview of the whole implemented planning subarchitecture, which is depicted in Figure 2. The heart of it is the TFD/M planner as described in Section 4. Its input is generated by the subcomponents depicted in the box at the bottom left – AWM and GM contribute the systems current state, which corresponds to the planner's initial state, and GoalGen transforms a user's command into a PDDL description of the goal.

As we described in the previous section, symbolic planning is not sufficient in a dynamic real-world environment. This is taken into account by the additional use of semantic attachments (depicted as Modules), which do complex calculations during planning such as checking spatial conditions for grasping or deciding where to place an object.

The last depicted component that is part of the planning subarchitecture is monitoring, which decides if the planner is triggered at all based on the success of the currently active plan's execution.

6 Conclusion

In this paper we have presented the planning system used in the DESIRE project. We have shown how the current state of the robot is generated from information of distributed subarchitectures and how additional knowledge is passed between consecutive planning episodes to avoid endless loops in continual planning that might arise if some action seems executable on the symbolic level but is indeed not in the real world.

Another difficulty arising when planning in dynamic and uncertain environments is the necessity for calculations that exceed the expressive power of PDDL. With the concept of semantic attachments we have presented a way to overcome this hindrance.

Acknowledgments This research was supported by the German Federal Ministry of Education and Research (BMBF) under grant no. 01IME01-ALU.

References

1. Plöger, P., Pervölz, K., Mies, C., Eyerich, P., Brenner, M., Nebel, B.: The DESIRE Service Robotics Initiative. Künstliche Intelligenz 08 (4), pp. 29–32 (2008)

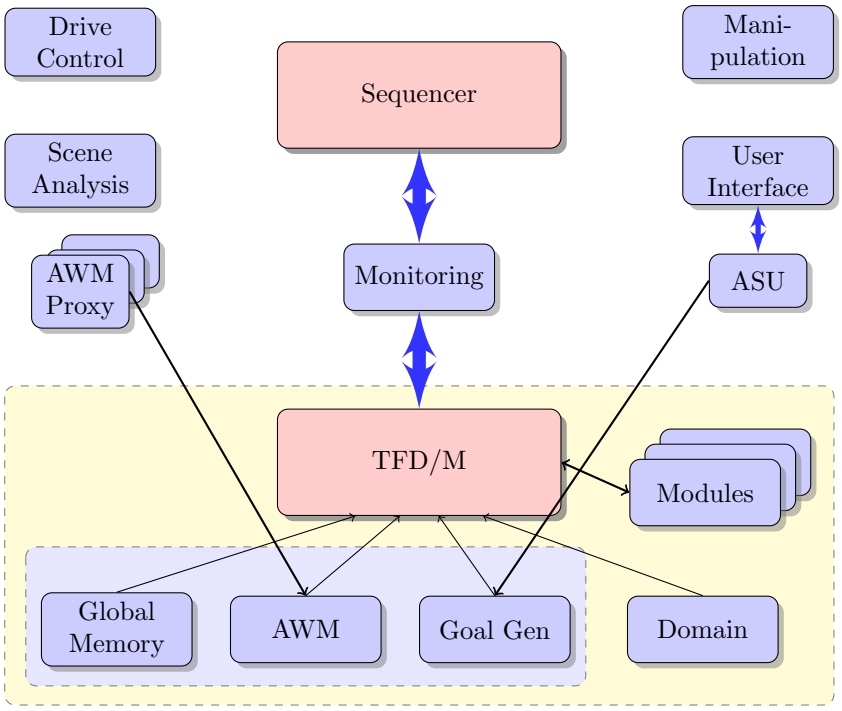

Fig. 2. System Architecture of the Planning subcomponent in DESIRE.

2. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practise. Morgan Kaufmann Publishers, San Fransisco, CA (2004)
3. Helmert, M.: The Fast Downward Planning System. JAIR (26), pp. 191-246 (2006)
4. Eyerich, P., Mattmüller, R., Röger, G.: Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. In: 19th Int. Conf. on Automatic Planning and Scheduling, pp. 130–137. AAAI Press, California (2009)
5. Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., Nebel, B.: Semantic Attachements for Domain-Independent Planning Systems. In: 19th Int. Conf. on Automatic Planning and Scheduling, pp. 114–121. AAAI Press, California (2009)
6. Brenner, M., Nebel, B.: Continual Planning and Acting in Dynamic Multiagent Environments. Journal of Autonomous Agents and Multiagent Systems 19 (3), pp. 297-331 (2009)
7. Littman, M. L., Goldsmith, J., Mundhenk,M.: The Computational Complexity of Probabilistic Planning. JAIR (9), pp. 1-36 (1998)
8. Rintanen, J.: Constructing conditional plans by a theorem-prover. JAIR (10), pp. 323352 (1999)
9. Hoffmann, J, Nebel, B.: The FF Planning System: Fast Plan Generation Through Heuristic Search. JAIR (14), pp. 253-302 (2001)
10. Geffner, H.: Functional STRIPS: a more flexible language for planning and problem solving. In Logic-Based Artificial Intelligence. Dordrecht, Holland: Kluwer (2000)
11. Fox, M., Long, D.: PDDL2.1: An extension to PDDL for expressing temporal planning domains. JAIR (20), pp. 61-124 (2003)