# Hierarchical Plan-based Robot Control in Open-Ended Environments

Dissertation

zur Erlangung des akademischen Grades
Dr. rer. nat
an der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg

eingereicht beim Fach-Promotionsausschuss Informatik von
Dominik Off
aus Hamburg (Deutschland)

Mai 2012

# Gutachterinnen/Gutachter

Prof. Dr. Jianwei Zhang
Prof. Bernd Neumann, PhD

Tag der Disputation: 15.03.2013

von der Prüfungskommission genehmigte korrigierte Fassung

# Abstract

Artificial agents need to plan their future course of action for the purpose of autonomously and flexibly performing tasks. State-of-the-art planning techniques can provide artificial agents to a certain degree with autonomy and robustness. However, previous planning approaches are typically limited by the fact that they are based on the assumptions that all relevant information is initially available and a complete plan can be generated in a single, monolithic process prior to executing any action. Comparatively little attention has been paid to the need for planning with incomplete, open-ended domain models that enable the reasoning about the active acquisition of relevant but missing information.

This thesis introduces a novel hierarchical planning approach that extends previous approaches by additionally considering decompositions that are only applicable with respect to a consistent extension of the (open-ended) domain model at hand. The introduced planning approach is integrated into a plan-based control architecture that interleaves planning and execution automatically so that missing information can be acquired by means of active knowledge acquisition. The plan-based control system can automatically determine what information is relevant for a task at hand as well as what information can be acquired by the corresponding agent. Whenever it is more reasonable, or even necessary, to acquire additional information prior to making the next planning decision, the planner postpones the overall planning process, and the execution of appropriate knowledge acquisition tasks is automatically integrated into the overall planning and execution process.

Real-world and simulation-based evaluation results demonstrate that this approach enables a physical robot agent to perform tasks even if no knowledge about the dynamic aspects of the environment is available a priori.

# Zusammenfassung

Künstliche Agenten müssen zukünftige Aktionen planen, um gegebene Aufgaben autonom und flexibel ausführen zu können. Bestehende Planungsansätze können genutzt werden, um künstliche Agenten zu einem gewissen Grad mit Autonomie und Robustheit auszustatten.

Typischerweise ist die Flexibilität von bestehenden Ansätzen durch die Annahmen beschränkt, dass alle relevanten Informationen von vornherein vorhanden sind und ein vollständiger Plan in einem einzigen, monolithischen Planungsprozess, vor der Ausführung von Aktionen, generiert werden kann.

Die Nutzung von offenen Domänenmodellen, die in der Lage sind, die aktive Akquirierung von relevanten Informationen zu berücksichtigen, ist bisher vergleichsweise wenig berücksichtigt worden.

Diese Arbeit beschreibt einen neuen hierarchischen Planungsansatz, der bestehende Ansätze erweitert, indem er zusätzlich hierarchische Zerlegungen berücksichtigt, die lediglich bezüglich einer konsistenten Erweiterung des aktuellen (offenen) Domänenmodells anwendbar sind. Der eingeführte Planungsansatz ist dergestalt in eine planungsbasierte Kontrollarchitektur integriert, dass sich die Generierung und Ausführung von (partiellen) Plänen abwechselt, sodass fehlende Informationen mittels aktiver Wissensakquirierung ermittelt werden können. Das planungsbasierte Kontrollsystem ist in der Lage automatisch zu erkennen, welche Informationen für die aktuelle Aufgabe relevant sind und wie diese Informationen ermittelt werden können. Der Planer verschiebt die weitere Zerlegung eines Plans auf einen späteren Zeitpunkt, wann immer es sinnvoller oder sogar notwendig ist, zusätzliche Informationen vor der Fortführung des Planungsprozesses zu ermitteln. Die Planung und Ausführung von entsprechenden Wissensakquirierungsaufgaben wird automatisch in den gesamten Ausführungs- und Planungsprozess integriert.

Experimente in einer realen und einer simulationsbasierten Umgebung demonstrieren, dass dieser Ansatz einen Roboter dazu befähigen kann, Aufgaben autonom auszuführen, obwohl eine Vielzahl von relevanten Informationen nicht a priori vorhanden ist.

# Contents

Chapter 1

# Introduction

*In my view, some of the more important directions for growth in the near future include planning in multiagent environments, reasoning about time, dynamic external information, acquiring domain knowledge, and cross-pollination with other fields.* (Nau, 2007)

## Contents



Intelligent agents (e.g., robots) are supposed to perform tasks autonomously. Instead of instructing agents by means of a detailed sequence of low-level actions, we want to instruct them via high-level tasks like "Bring me a cup of coffee". In other words, we want to tell robots *what* to do, but *not how* to do it. This objective makes it necessary for agents to plan flexibly how they perform desired tasks in a given state of the world. For example, if we instruct a robot to cross a door which is open, then it can simply approach and cross the door. However, if the door is closed, then the robot first has to open the door. The situation is even more complicated if the robot holds an object in its hands. In this case, it must first put the object aside, open the door, grasp the object again, and then it can pass through the door. This simple example makes it apparent that how a task can actually be performed heavily depends on the concrete situation. In order to deal with this dependency, artificial agents have to flexibly plan their future course of action. Thus, they require strong reasoning and planning capabilities. Unfortunately, generating and executing plans that perform tasks in unstructured, real-world environments is for several reasons a difficult—often unsolvable—problem for artificial agents. One important reason is

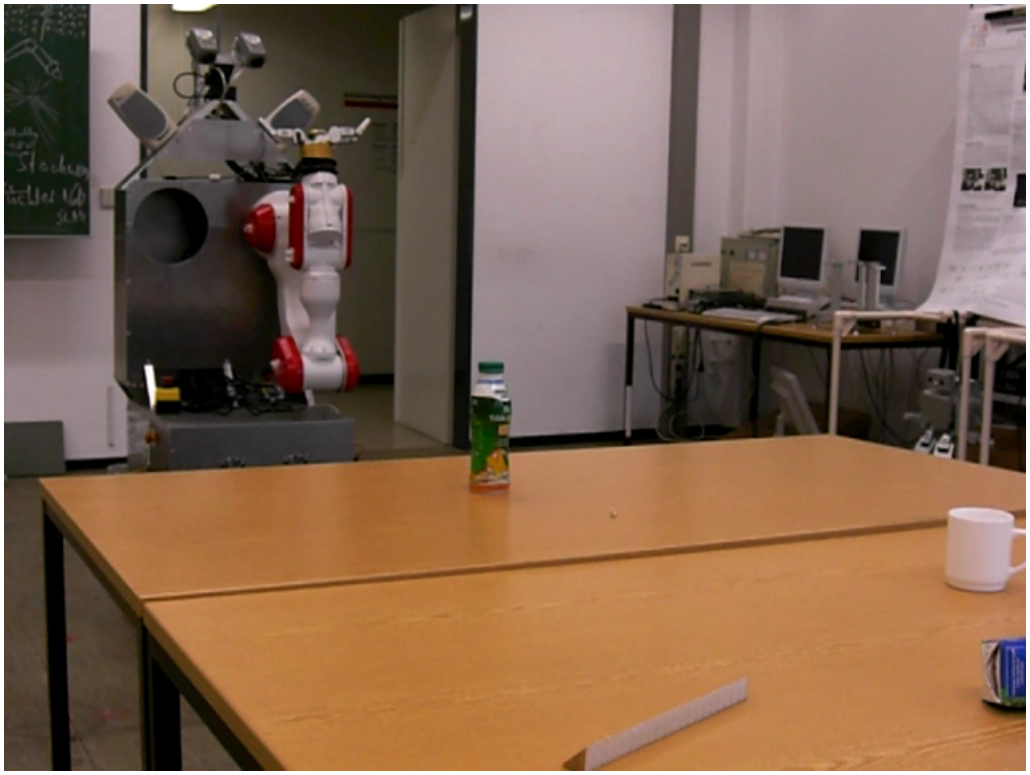


Figure 1.1.: Illustration of a situation where a mobile service robot is instructed to pick up an object from a table.

the fact that it is impossible to a priori equip agents with all relevant knowledge. Dealing with this lack of knowledge is not extensively considered by previous work on AI planning and plan-based control, though seen as an important direction for further growth (Nau, 2007).

An *open-ended domain model* is defined as a domain model from which an agent can in general neither derive all information nor all possible states (e.g., all objects) of the world it inhabits. The goal of this work is to develop a plan-based control system that enables autonomous agents to perform tasks reasonably in unstructured, real-world environments even if they only have an incomplete, open-ended model of such an environment. In particular, adequate knowledge representation, reasoning and planning approaches are intended to be developed and composed to a coherent control architecture.

## 1.1. Motivating Example

Let us consider the situation illustrated by Figure 1.1, where a mobile service robot is instructed to pick up an object from a table. Picking up an object from a table is a typical task for a service robot. Such a task has been successfully executed by a multitude of service robot systems. It does not seem to be very complicated.

The situation becomes more difficult if the robot is not only supposed to be able to pick up an object in a specific situation (e.g., for a certain position of the object on the table), but supposed to be able to perform the task in as many concrete situations as possible. In this case, we cannot define a sequence of low-level actions that always constitutes a valid plan for the task of picking up an object from a table. The robotic agent has to be able to flexibly plan its future course of action so that it can pick up the object in a large set of concrete situations. In other words, we need to somehow integrate a planning system into the control architecture of the autonomous agent. However, for real-world domains like robotics, an agent usually only has an incomplete, open-ended model of its environment. Thus, a lot of relevant information is often not available at planning time. This rules out the application of most of the previous planning approaches, since they are based on the assumption that a planner is a priori equipped with all relevant information and a plan can be generated in a single, monolithic process (Nau, 2007). Comparatively less attention has been paid to task planning in real-world situations where the model of the environment is inherently open-ended.

However, if previous work on AI planning turns out to be too inflexible to enable a service robot to perform a typical task like picking up an object from a table, then the question is: How can we enable a robotic agent to perform such a task in a realistic situation where not all relevant information is available a priori? For example, how can a robot plan to pick up an object from a table if the position of the object is unknown, or if it has no knowledge about objects on the ground that can obstruct a passage? How can the robot figure out what relevant information is missing? And how can the required information be acquired? All these questions have not been sufficiently addressed by previous work on plan-based control.

## 1.2. Objectives

The previous section illustrated that previous work on AI planning and plan-based control often fails to enable artificial agents to perform tasks autonomously in real-world environments. Existing AI planning systems can enable artificial agents—at least to some degree—to autonomously plan their future course of action. Planning algorithms have been developed that in principle are efficient enough to solve large planning problems in real time. However, classical planning approaches fail to generate plans when necessary information is not available at planning time, because they rely on having a complete representation of the current state of the world (Nau, 2007).

*Conformant*, *contingent*, or *probabilistic* planning approaches can be used to generate plans in situations where insufficient information is available at planning time (Russell and Norvig, 2010; Ghallab et al., 2004). These approaches typically generate conditional plans—or policies—for all possible contingencies. Conformant, contingent, or probabilistic planning approaches extend classical approaches by allowing for generated plans to contain conditional branches for (all) possible contingencies.

The branches are attempted to be resolved during execution according to the observed situation. Nevertheless, these approaches assume, like classical planning, that a—possibly conditional—complete plan can be generated in a single, monolithic planning process. This assumption renders conformant, contingent or probabilistic planning approaches intractable for real-world situations, since the number of contingencies is too large (possibly infinite), while opening doors for planning approaches that can more flexibly interleave planning and execution.

A more promising approach for agents that act in open-ended domains is *continual planning* (Brenner and Nebel, 2009), which enables the interleaving of planning and execution so that missing information can be acquired by means of active information gathering. Thus, continual planning approaches do not presume that the planning process is monolithic. Existing continual planning systems can deal with incomplete information, but their reasoning capabilities are usually limited by the fact that they are based on classical planning systems that do not natively support open-ended domain models. They typically rely on the assumption that all possible states of a domain are known (i.e., they assume so-called *bounded incompleteness*). This makes it, for example, difficult to deal with a priori unknown object instances. Another important issue that is not directly considered by previous work is the fact that a knowledge acquisition task $task_1$ can—like any other task—make the execution of an additional knowledge acquisition task $task_2$ necessary which might require the execution of the knowledge acquisition task $task_3$ and so on. Consider, for example, a situation where a robot is instructed to deliver Bob's mug into Bob's office. Moreover, let us assume that the robot does know that Bob's mug is in the kitchen, but does not know the exact location of the mug. In this situation, the robot needs to perform a knowledge acquisition task that determines the exact location of Bob's mug. However, in order to do that via perception the robot first needs to go into the kitchen. If the robot does not have all necessary information in order to plan to get into the kitchen (e.g., it is unknown whether the kitchen door is open or closed), then it needs to first perform additional knowledge acquisition tasks that acquire this information. Existing continual planning approaches usually fail to cope with such a situation. In contrast, the continual planning and acting approach proposed in this work is intended to deal with these kinds of situations and thus can enable artificial agents to perform tasks in a significantly larger set of situations.

Furthermore, previous continual planning approaches never attempted to exploit the domain specific knowledge encoded in the domain model of a *Hierarchical Task Network* (HTN) planner. The domain specific knowledge of an HTN planner is typically used to more efficiently solve classical planning problems and generalizations thereof. However, in addition to the usual usage, this knowledge has the potential to significantly contribute to the efficient answering of the following questions, that are essential for continual planning systems: What information is relevant for the purpose of finding a plan for the task at hand? What information implies which additional ways to continue the planning process? Moreover, hierarchical planning

is particularly interesting for continual planning approaches that generate partial plans, since hierarchical planners can naturally represent partial plans by means of leaving some parts of the plan hierarchy unexpanded.

The main objective of this thesis is to propose a hierarchical-planning-based control system that can deal with real-world situations more flexibly where an agent only has an open-ended model of the world it inhabits. The plan-based control system is intended to be a general-purpose framework that can in principle be used to control various hard- and software agents. Its application is not limited to physical robots, though it is intended to be particularly able to deal with some of the challenges real-world systems are usually confronted with (e.g., the open-endedness). The system is supposed to be capable of automatically detecting missing but relevant information. Whenever reasonable, the hierarchical planning process should be deferred, and missing information should be acquired automatically.

The primary challenges in implementing such an approach is threefold:

1. AI planning approaches are equipped with a model of a situation and the activities that can be performed by a corresponding agent. Such a model is usually called the *domain model* (McCluskey, 2002). The domain model of AI planning systems is typically based on the assumption that all relevant information is available at the beginning of the planing process. Unfortunately, assuming that all information is a priori available makes planning impossible for all real-world situations where an agent only has an incomplete, open-ended model of its environment. Comparatively little attention has been paid to the need for open-ended domain models that feature the reasoning about active knowledge acquisition of relevant but missing information. The first objective of this work is to developed such an open-ended domain model. It is intended to provide the basic knowledge representation and reasoning capabilities for the overall plan-based control system.

2. HTN planning has been successfully applied to a variety of applications. Nevertheless, previous HTN planning approaches are—like classical planning—typically limited by the fact that they are based on the assumption that all necessary information is initially available and a complete plan can be generated in a monolithic process prior to executing any action. The second aim of this thesis is to develop an extended HTN planning approach that is capable of dealing with an open-ended domain model by means of additionally considering ways of performing tasks that are only possible with respect to a consistent extension of the domain model at hand (i.e., require the acquisition of additional information). This work attempts to investigate if and how the additional knowledge encoded in the domain model of an HTN planner can be exploited to deal with an essential part of continual planning approaches: reasoning about relevant extensions of the domain model at hand and active knowledge acquisition.

3. The third objective of this thesis it to develop a plan-based control system that integrates the proposed HTN planning approach so that planning and execution is interleaved, and missing information can be acquired by means of active knowledge acquisition. The plan-based control system is intended to be capable of automatically determining what information is missing but relevant for the planning process. If it is more reasonable to acquire additional information prior to continuing the planning process for the task at hand, then the planning process should be deferred. The control system is supposed to automatically create corresponding knowledge acquisition tasks for missing information. The planning and execution of these knowledge acquisition tasks is integrated into the overall planning and execution process. Then, the deferred planning process is continued after additional information has been acquired. In this way, the plan-based control system is intended to provide the planning, reasoning, and high-level control methods that enable an autonomous agent to automatically perform tasks, though it only has an incomplete, open-ended model of the world it inhabits.

The ultimate aim of this line of work is to enable intelligent agents that exhibit goal-oriented, high-quality behaviors in unstructured, real-world environments. This thesis aims at taking a step into this direction by means of providing a new plan-based control approach that can more flexibly deal with the open-endedness of real-world environments and the models thereof.

## 1.3. Natural Cognitive Systems: A Source of Inspiration

Human beings are probably the most elaborated natural cognitive systems. They appear to make heavy use of various reasoning and planning techniques to generate plans for the tasks they carry out on a day-to-day basis. The investigation of how humans usually solve the every-day task of making a cherry pie serves as a source of inspiration for the objectives and the general approach of this work.

So let us assume that we are confronted with the task of making a cherry pie, while not knowing how this actually can be done. How would we solve that problem? Or in other words: How would we generate and execute a reasonable plan in order to perform the task of making a cherry pie? First, we would probably look for an adequate recipe that describes how the desired cake can be made. From a more technical perspective of an AI researcher, a recipe can be viewed as domain knowledge that encodes how the task of making a cake can be performed. If we want to generate a plan based on the recipe, then we usually need information about dynamic facts (e.g., the availability and location of necessary ingredients or tools). The availability and location of necessary ingredients and tools can be seen as a precondition. For the purpose of correctly making the cake, it is essential to ensure that this precondition is fulfilled. It can be ensured that the precondition

holds by either finding out that it holds in the current state of the world or by actively making it true. However, first of all, we have to find out what part of the precondition is already fulfilled and what part does not hold in the current situation. Unfortunately, in a real-world environment, it is often impossible to determine whether a precondition—or any other statement—holds or cannot hold only by reasoning, since this would require us to have sufficient information about the current state of the world. Thus, we are often in the situation where we just do not know whether a statement is true. Nevertheless, even in this situation our reasoning capabilities are able to analyze statements in a more fine-grained manner. The more fine-grained analysis results in partitioning statements into a subpart that is known to be true and a subpart that is neither known to be true nor known to be false. In our making a cherry pie example, we might know that we have all necessary tools, but are unsure about the availability of the necessary ingredients. In this situation, the following questions might have to be answered: Are all necessary ingredients available? Is an egg available? Where are the eggs located? Where and how to get unavailable ingredients? Which of these questions have to be answered depends on the concrete circumstances; thus, it is hard to a priori define when to acquire what information. For example, if it can be determined that all necessary ingredients are available, then the question of where and how to get missing ingredients does not have to be considered. Hence, the knowledge acquisition process heavily depends on the concrete situation. We should therefore keep in mind that it is desirable that artificial agents can generate questions and corresponding knowledge acquisition tasks in a generic manner, since the explicit representation of when to acquire what information is practically unfeasible due to the sheer magnitude of possible situations.

In many cases, there are several ways to answer a question. For example, if we want to known whether an egg is available, then we can either ask our roommate, or we can take a look into the fridge. In this situation, our roommate and the visual sensing modality serve as *external knowledge sources* that enable us to extend our knowledge about the environment. The answering of generated questions can also be viewed as a task for which a plan has to be computed and executed. For example, we have to generate and execute a plan for the purpose of determining whether sufficient eggs are available. Finally, the result of the knowledge acquisition process has to be integrated into the overall planning and executing process.

The overall example demonstrates that we—as human beings—usually are able to perform tasks even if not all necessary information is a priori available by:

1. partitioning preconditions into a subpart that is known to be true and a subpart that is indeterminable;

2. generating questions about the indeterminable part of a precondition;

3. generating, scheduling, and executing plans in order to acquire the necessary information by submitting the queries to external knowledge sources;

4. and reasonably integrating acquired information into the interleaved planning and execution process.

These observed problem solving strategies serve as an important source of inspiration for the overall approach of this work.

## 1.4. Outline of the Thesis

The dissertation begins by means of outlining the way to a novel hierarchical planning approach tailor-made for open-ended domains. Chapter 2 is divided into three sections. Section 2.1 briefly describes the idea of state-of-the-art HTN planning, Section 2.2 outlines the general idea of the novel HTN planning approach, and Section 2.3 itemizes the requirements for such a planning approach.

The Chapters 3, 4, 5, 6 all conclude with a brief summary and discussion of related work.

Chapters 3 and 4 describe a novel open-ended domain model.

Chapter 3 begins by proposing a new, open-ended state model. The state model represents knowledge about the state of world. It features the reasoning of consistent extensions of the state model at hand. The novel notion of a *possibly-derivable statement* enables the reasoning about extensions with respect to a certain statement (e.g., the precondition of an HTN method). The corresponding reasoning process is central for the novel HTN planning approach outlined in Chapter 2 and more precisely described in Chapter 5. The state model is represented based on *definite clauses* such that the overall model constitutes a *definite program*. Therefore, the process of deriving information from the state model can be reduced to the well-know process of deriving information from the corresponding definite program, and high performance definite clause reasoners (e.g., Prolog systems) can be used to easily and efficiently implement the reasoning process. The basic state model is extended by additional concept relations that make it possible to improve the performance by means of exploiting the knowledge encoded by these relations. The performance of the main reasoning process is experimentally compared with a typical closed-world implementation.

Chapter 4 describes the *activities model* that models the activities which can be performed by the agent itself. It introduces a special kind of task, namely a *knowledge acquisition task*. HTN methods for knowledge acquisition tasks describe what knowledge acquisitions are possible under what conditions, how expensive it is to acquire information from a specific knowledge source, and how a knowledge acquisition task can be performed. The notion of an *applicable* planning operator or HTN method is extended to the notion of a *possibly-applicable* planning operator or HTN method. Furthermore, it is described how probabilistic information can be integrated into the underlying cost model so that a planner can follow the principle of *maximum expected utility*. Additionally, the novel concept of *high-level percepts* is

introduced. High-level percepts represent multimodal integration processes. They describe how a set of low-level percepts can be mapped to a single high-level percept.

Chapter 5 introduces the principled new HTN planner ACogPlan. It describes how the proposed domain model can be exploited so that the requirements described in Section 2.3 can be fulfilled. Moreover, it describes the algorithm of the planner as well as discusses the completeness and soundness thereof.

Chapter 6 proposes a novel plan-based control system. The ACogPlan planner is integrated into the control system so that planning and execution is interleaved. The chapter describes the algorithm of the plan-based controller. An example for the complete execution process of a task illustrates the functional interaction between the plan-based controller and the planner.

Chapter 7 describes how one can represent a domain model for a typical service robotic domain, whereby the goal of this chapter is twofold. First, it is intended to demonstrate how some of the described features of the ACogControl framework can be used to represent knowledge for a typical service robotic domain. In particular, the goal is to point out that the proposed architecture can release a domain engineer from the burden of explicitly dealing with the fact that often not all relevant information is available at the beginning of the planning and execution process. Second, the chapter describes a representative subset of the domain model used for the experimental evaluation with a physical service robot described in Chapter 9.

Chapter 8 describes the hard- and software components of the mobile service robot TASER that was used as an experimental platform for the evaluation of the proposed control system.

Chapter 9 describes real-world and simulation-based experiments and discusses the results thereof.

Finally, Chapter 10 concludes with a brief summary of the main contributions of this work and discusses possible directions for future work.

Chapter 2

# Towards HTN Planning in Open Ended-Domains

*In most automated-planning research, the information available is assumed to be static, and the planner starts with all of the information it needs. In real-world planning, planners may need to acquire information from an information source such as a web service, during planning and execution. This raises questions such as What information to look for? Where to get it? How to deal with lag time and information volatility? What if the query for information causes changes in the world? If the planner does not have enough information to infer all of the possible outcomes of the planned actions, or if the plans must be generated in real time, then it may not be feasible to generate the entire plan in advance. Instead, it may be necessary to interleave planning and plan execution.* (Nau, 2007)

## Contents



This chapter describes the general idea of and the requirements for hierarchical planning in open-ended domains. It attempts to provide a first informal motivation and introduction of the proposed HTN planning approach, that extends previous approaches in order to be able to deal with incomplete, open-ended domain models.

A more detailed formalization and description of the domain model, the underlying reasoner, and the planning algorithm will be provided in Chapter 3, 4 and 5.

Section 2.1 briefly describes state-of-the-art HTN planning using a simple example with a robotic agent. The general idea of the new HTN planning approach for open-ended domains is introduced in Section 2.2. A planning system that implements this idea should fulfill a set of requirements. These requirements are discussed in Section 2.3.

## 2.1. State-of-the-Art HTN Planning

This section provides a brief informal introduction to HTN planning. A more comprehensive introduction to HTN planning can be found in (Ghallab et al., 2004).

### 2.1.1. Introduction

The history of HTN planning goes back to first HTN planner NOAH (Sacerdoti, 1975). This approach has been succeeded by a multitude of other HTN planners including NONLIN (Tate, 1977), SIPE (Wilkins, 1983), O-Plan (Drummond and Currie, 1989), SIPE-2 (Wilkins, 1990), UMCP (Erol et al., 1994), DPOCL (Young et al., 1994), AbNLP (Fox and Long, 1995), SHOP (Nau et al., 1999), and SHOP2 (Nau et al., 2003). All these approaches have in common that they use hierarchical knowledge to more efficiently solve classical planning problems and generalizations thereof. Like classical planners, the approaches are typically based on the assumption that all relevant information is available at the beginning of the planning process, and a given planning problem can be solved in a single, monolithic planning process.

In contrast to classical planning, the goal of HTN planning is not to achieve a set of goals, but to perform a set of tasks. Planning proceeds by means of successively decomposing tasks into a number of subtasks until the level of *primitive tasks* is reached. Primitive tasks are tasks that can directly (i.e, without additional task planning) be executed by an agent. Like classical planners, an HTN planner has knowledge in the form of *planning operators* about what action primitives can be performed and how the action primitives effect the state of the world. In addition to knowledge about primitive actions, an HTN planner has access to a set of *HTN methods* that prescribe how a task can be decomposed into a sequence of subtasks. HTN methods guide the planning process. They constitute the set of valid ways to decompose a task into a number of subtasks. The planner only considers plans that result from the successive application of a valid decomposition. In this way, the domain specific knowledge encoded in HTN methods can prune the search space and enable an HTN planner to find plans significantly faster than classical planning systems.

```
operator(approach(Entity),
  % precondition
  true,
  % delete-set
  [approached(_)],
  % add-set
  [approached(Entity)]).
operator(cross(Door),
  % precondition
  (approached(Door) ∧ in_room(R) ∧ connect(R,Door,R2)),
  % delete-set
  [approached(Door),in_room(R)],
  % add-set
  [in_room(R2)]).
operator(pick_up(Obj,Table),
  % precondition
  (on(Obj,Table) ∧ approached(Table) ∧ free(hand)),
  % delete-set
  [on(Obj,Table),free(hand)],
  % add-set
  [in_hand(Obj)]).
operator(put_down(Obj,Table),
  % precondition
  (approached(Table) ∧ in_hand(Obj)),
  % delete-set
  [in_hand(Obj)]).
  % add-set
  [on(Obj,Table),free(hand)],
```

Figure 2.1.: Operators for the illustrative example.

### 2.1.2. A Running Example

Let us consider a very simple example for the purpose of explaining the general idea of HTN planning. Imagine a robotic agent that can perform four primitive actions. It can approach known entities (e.g., a table or a door) via the action `approach(Entity)`, it can cross doors via the action `cross(Door)`, it can pick up objects from a table via the action `pick_up(Obj,Table)`, and it can put objects down on a table via the action `put_down(Obj,Table)`. The agent has knowledge about the actions it can perform and about the current state of the world. This knowledge is called the *domain model* (McCluskey, 2002). The domain model is represented by the *domain specification*. From the perspective of automated planning, the actions of an agent constitute the basic operators that are considered by a planner. Therefore, the domain specific model that encapsulates knowledge about the basic actions that can be performed by an agent is called a *planning operator* (Ghallab et al.,

```
method(deliver(Obj,Table),
  % precondition
  (on(Obj,Table1) ∧ in_room(Table,R))
  % subtasks
  [approach(Table1),pick_up(Obj,Table1),
    move_to(R),approach(Table),put_down(Obj,Table)]).
method(move_to(R),
  % precondition
  (in_room(agent,R1) ∧ connect(R1,Door,R) ∧ open(Door))
  % subtasks
  [approach(Door),cross(Door)]).
```

Figure 2.2.: Methods for the illustrative pick-up and delivery task.

2004). For each of the four aforementioned actions that the robotic agent can perform, the domain specification contains a planning operator. The specification of these planning operators is shown in Figure 2.1. The domain specification is defined in Prolog-like syntax (Deransart et al., 1996). Departing from the Prolog syntax, preconditions are defined using the connectives '$\wedge$' for conjunction and '$\vee$' for disjunction. For the domain specifications presented in this work, keywords are bold and variables are highlighted blue.

A more precise definition of the planning operator model used for the proposed planning system is described in Section 4.2.2. For the context of this example, a planning operator is specified by an atomic formula of the following form:

`operator(Name,Precondition,Delete-Set,Add-Set)`

For such an operator specification, the term `Name` denotes the name of the operator. Like for classical planning, each planning operator has a precondition. For example, in order to perform the action `pick_up(Obj,Table)`, the robot must have a free hand, it must have approached the table and the object must actually be on that table. The precondition is represented by a logical statement denoted by the term `Precondition`. Furthermore, `Delete-Set` denotes a set of literals that do not hold and `Add-Set` denotes a set of literals that hold after the execution of the action. In this way, these sets define the effect of the operator. For example, consider the planning operator `pick_up(Obj,Table)` shown in Figure 2.1. After the robot has picked up an object from a table, the hand of the robot is not free, and the object is not on the table anymore. Thus, the literals `free(hand)` and `on(Obj,Table)` are removed from the domain model of the robot. In contrast, `in_hand(Obj)` is added to the domain model, since the robot will have the object in its hand after it has performed the corresponding pick-up action.

In addition to the definition of planning operators, the domain specification contains two HTN methods shown in Figure 2.2. A more precise definition of the HTN method model used for the proposed planning system is described in Section 4.2.3.

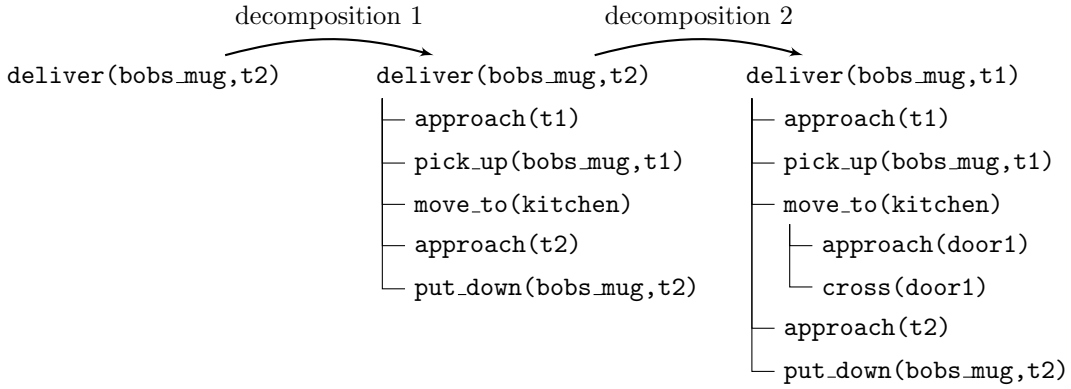



Figure 2.3.: Illustration of the planning process for the task of delivering Bob's mug to table `t2`.

For the context of this example, HTN methods are represented by atomic formulas of the following form:

$$\texttt{method}(\texttt{Task}, \texttt{Precondition}, \texttt{Subtasks})$$

For such a method, the term `Task` denotes the task for which the method describes a possible decomposition, `Precondition` is a logical statement that serves as the precondition, and `Subtasks` is a sequence of subtasks in which the task can be decomposed.

The first method shown in Figure 2.2 defines that the task of delivering an object to a certain table can be decomposed into the following tasks: approach the table on which the object currently is located, pick up the object, move to the room in which the goal table is, approach the goal table, and put the object down. The second method describes how a robot can move to a different room in a situation where the room is directly connected via a door to the room the robot is currently located in. It is assumed that the knowledge of the robot about its environment is represented by a set of literals that hold in the current situation. For this example, let us assume that the robot believes that the following literals hold:

```
in_room(agent,lab), in_room(t1,lab), in_room(t2,kitchen),
on(bobs_mug,t1), free(hand), connect(lab,door1,kitchen),
connect(lab,door2,kitchen), open(door1).
```

For this example, an HTN planner generates a plan via performing two decompositions as illustrated by Figure 2.3. It first decomposes the task `deliver(bobs_mug,t2)` into the five subtasks defined by the corresponding method. Except for the subtask `move_to(kitchen)`, all subtasks are primitive and do not need to be further decomposed. Subsequently, the planner decomposes the task of moving into the kitchen into the task of approaching and crossing `door1`. What decompositions are possible

depends on available HTN methods that are relevant and the knowledge the agent has about the state of the world. More precisely, decomposing a task into a number of subtasks consists of the following three steps:

1. Choose an HTN method that is *relevant* (i.e., describes how this task can be decomposed) for a task.
2. Try to generate an (valid) instance of the chosen HTN method.
3. Decompose the task as specified by an instance of the chosen HTN method.

For example, let us take a closer look at the second decomposition illustrated by Figure 2.3. The planner tries to generate a plan for the task of moving into the kitchen. The planner chooses the second method shown in Figure 2.2, since this is the only method that is relevant for the task `move_to(kitchen)`. As a first step of the instantiation process, the variable `R` of the method specification needs to be replaced with the constant `kitchen` so that the method is actually relevant for the task `move_to(kitchen)`. In order to find a valid instance of the method, the planner forwards the (partially instantiated) precondition of the method to a reasoning system. It is not important here whether this reasoner is tightly integrated with the planner or rather an external component. Based on the knowledge about the current state of the world that is encapsulated by the domain model, the reasoner tries to derive an instance of the precondition. In this case, the reasoner needs to derive an instance of the following precondition:

$$\texttt{in\_room(agent,R1)} \wedge \texttt{connect(R1,Door,kitchen)} \wedge \texttt{open(Door)}$$

In other words, the reasoner tries to find a door that connects the room in which the agent is located in with the kitchen and is open. In this example, the reasoner can deduce from the domain model that the agent is in the lab, that `door1` connects the lab with the kitchen, and that `door1` is open. Thus, it can derive the following instance of the precondition:

$$\texttt{in\_room(agent,lab)} \wedge \texttt{connect(lab,door1,kitchen)} \wedge \texttt{open(door1)}$$

This instance is the results of applying a substitution to the precondition of the method that replaces `R` with `kitchen`, `R1` with `lab`, and `Door` with `door1`. The instantiation process applies the same substitution also to the sequence of subtasks. For the example at hand, this means that the definition of the subtasks is transformed to the following sequence by means of replacing `Door` with `door1`:

$$\texttt{[approach(door1),cross(door1)]}$$

Now the planner can decompose the task of moving to the kitchen into the primitive tasks of approaching and crossing `door1`. How many instances of a method can

be generated depends on how many instances of its precondition can be derived by the underlying reasoning systems. Every instance of the precondition that can be derived results in a corresponding instance of the method.

## 2.2. HTN Planning in Open-Ended Domains: General Idea

HTN planning is one of the most application-oriented AI-planning approaches (Nau, 2007). It has been used in a variety of applications including robotics (Weser et al., 2010; Hartanto, 2009). Furthermore, HTN planning is known to have good performance characteristics. For example, the good performance characteristics of the HTN planning system SHOP2 (Nau et al., 2003) have been demonstrated at the *International Planning Competition* 2002 (Long and Fox, 2003).

Nevertheless, existing HTN planning approaches are—like classical planning–based on the assumption that a planner starts with all information it needs. Usually these systems are based on the *closed world assumption* (CWA). The CWA leads to the assumption that if an atomic fact $p$ cannot be deduced from a given knowledge base $KB$, then $\neg p$ can be assumed (Brachman and Levesque, 2004). Based on this implication, a knowledge base $KB$ is extended to a $KB^+$ in the following way:

$$KB^+ = KB \cup \{\neg p \,|\, p \text{ is atomic and } KB \not\models p\}$$

For instance, for the pick-up and deliver scenario described in Section 2.1, existing HTN planners usually assume that `door2` is closed, since the fact that `door2` is open is not part of the domain model (i.e., the knowledge base).

However, for many real-word situations, agents act in an *open-ended domain*. An open-ended domain is defined as a domain in which an agent can in general neither be sure to have all information nor know all possible states (e.g., all objects) of the world it inhabits. In open-ended domains, often not all necessary information is available in order to generate a complete plan or make a reasonable planning decision. For example, a robot cannot generate a complete plan for the task of picking up an object from a certain table if the exact location of the object is unknown. Hence, the following question arises: How can HTN planning be extended such that it can deal with open-ended domains?

As already described, HTN planning proceeds by means of successively choosing an instance of a relevant HTN method or planning operator for which an instance of the precondition can be derived with respect to the domain model at hand. In open-ended domains, however, it will often be possible to instantiate additional HTN methods or planning operators (i.e., which precondition is not derivable) if additional information is available. The general idea of the proposed HTN planning system is to also consider instances of relevant HTN methods and planning operators for which the precondition cannot be derived with respect to the domain model at hand, but is derivable with respect to a consistent extension thereof. To put it another

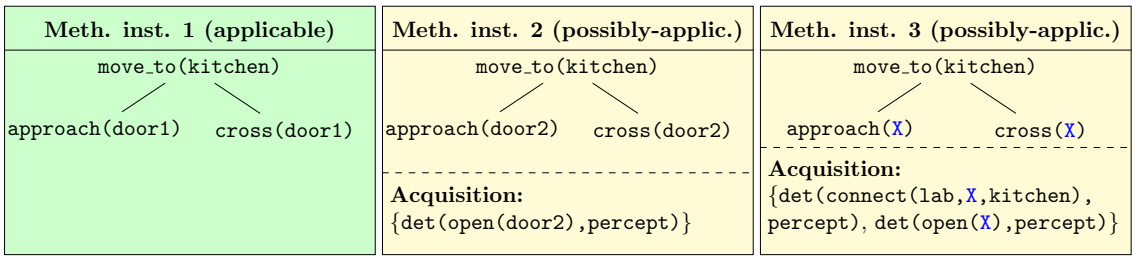



Figure 2.4.: Applicable and possibly-applicable method instances for the task `move_to(kitchen)`.

way, the planner additionally considers decompositions that are only applicable if additional information is available.

Let us consider the example described in Section 2.1.2 again. It has been described how an instance of the method for the task `move_to(kitchen)` can be generated that describes to move to the kitchen via `door1`. Previous HTN planning systems are unable to consider additional ways to move into the kitchen, since they only consider plans that are known to be executable with respect to the knowledge they have about the environment. However, in open-ended domains, it is possible to consider two additional instances of the relevant HTN method which cannot directly be applied, but are applicable in a consistent extension of the given domain model. HTN methods or planning operators that are only applicable with respect to an extension of an agent's domain model are called *possibly-applicable*. For example, it will also be possible to cross `door2` if the robot can find out that this door is open. Moreover, in open-ended domains, it can also be possible that there is another door which connects the lab and the kitchen. If the robot finds such a door that is open, then it can also move into the kitchen via this door. All the three possible ways to decompose the task `move_to(kitchen)` into a sequence of subtasks are illustrated by Figure 2.4.

Additionally considering possibly-applicable HTN methods or planning operators is important in situations where one cannot assume that all information is available at the beginning of the planning process. It often enables the generation—and execution—of additional plans. In particular, it can enable a planner to generate plans where it would otherwise be impossible to generate any plan at all. For example, if it were unknown whether `door1` is open or closed, then there would only be possibly-applicable method instances. Hence, without considering possibly-applicable method instances, a planner would fail to generate a plan for the task `move_to(kitchen)`, and the agent would be unable to achieve its goals. Moreover, if the optimal plan requires knowledge acquisition, then the optimal plan can only be found if possibly-applicable method and planning operator instances are considered. In other words, one can also benefit from the proposed approach in situations where it is possible to generate a complete plan without acquiring additional information.

## 2.3. Requirements

A planner that wants to consider possibly-applicable HTN methods or planning operators needs to be able to reason about extensions of its domain model. However, it is not reasonable to consider all consistent extensions of a domain model. The planner should only consider domain model extensions that fulfill the following three conditions:

1. A domain model extension should be *relevant* with respect to the overall tasks. An extension of a domain model is called *relevant* iff it implies an additional way to continue the planning process. For instance, a domain model extension that includes additional information about today's TV program is not relevant for the aforementioned task of moving into the kitchen, since it does not help the planner to find additional plans for this task. In contrast, if an extension of the domain model additionally includes the information that `door2` is open, then the planner can generate an additional plan for the task of moving into the kitchen. Hence, such an extension would be called relevant.

2. A domain model extension should be *possible* with respect to the domain model at hand. An open-ended domain can have an infinite number of consistent extensions. The challenge here is to enable a reasoner to exploit domain knowledge in order to rule out as many extensions as possible. For instance, we intuitively know that an object cannot be in two different rooms at once. Thus, we know that if Bob's mug is in the lab, then it is impossible that the same mug is also in the kitchen. Therefore, in such a situation, it would be beneficial if a reasoner does not consider extensions that include the fact that Bob's mug is in a different room.

3. A domain model extension should be *acquirable* with respect to a situation and the knowledge acquisition skills of the agent. It is unreasonable to consider extensions that are not acquirable by the corresponding agent. For example, if a robot is not able to determine whether a door is open or closed, then the second and third method instances shown in Figure 2.4 do not have to be considered.

The following part of this work is going to introduce a domain model, a reasoner, and a planning approach that fulfill these constraints.

Chapter 3

# Open-Ended State Model: Reasoning About An Open World

*Nevertheless, there are also many applications requiring open world axiomatizations. The most obvious of these is robotics; in such settings, it is naive to expect that a robot will have complete information about the initial state of its world, for example, which doors are open and which closed. Indeed, in the presence of complete information about its world, it is not clear why a robot would ever need sensors.* (Reiter, 2001)

## Contents

An agent that is supposed to perform tasks in a goal-oriented manner needs to plan its future course of actions. For the purpose of making reasonable planning decisions, an agent must have a model of its environment. In the AI-planning community, the model of the environment is often called the *domain model* (McCluskey, 2002). The domain model includes knowledge about the environment as well as knowledge about the activities (e.g., primitive actions) that can performed by the agent. In the context of this work, the former part of the domain model is called the *state model*, and the latter part is called the *activities model*. The state model and corresponding reasoning processes are described in this chapter, whereas the activities model is described in Chapter 4. The overall domain model presented in this work is called *ACogDM (Artificial Cognitive systems Domain Model)*.

The underlying state models of AI-planning approaches often rely on the assumption that all necessary information is available at the beginning of the planning process (Nau, 2007). Thus, they are assumed to be complete. Unfortunately, this presumption is in conflict with the fact that real-world agents act in dynamic, unstructured environments, where they can in general neither be sure of having all information nor of knowing all possible states (e.g., all objects) of the world they inhabit. Therefore, the model they have of the state of the environment is open-ended.

In contrast to most of the previous work, the state model described in this chapter is inherently open-ended and particularly suitable for real-world agents that inhabit an open world. The state model provides the basic reasoning capabilities for the planner briefly outlined in Section 2.2 and more comprehensively presented in Chapter 5. It enables the planner to reason about extensions of the state model at hand, and thus is an essential component of the overall plan-based control system developed in the context of this work.

A preliminary version of the state model has been presented in Off and Zhang (2011c).

## 3.1. Preliminaries

*Definite clauses* are used as the representational basement for state models. The definition and notation of *definite clauses*, *definite goals*, *definite programs* and *substitutions* is borrowed from Nilsson and Maluszynski (1995) and briefly introduced in this section.

A *substitution* is defined as a finite set of pairs $\{X_1/t_1, \ldots, X_n/t_n\}$ where each $t_i$ is a term and each $X_i$ a variable such that $X_i \neq t_i$ and $X_i \neq X_j$ if $i \neq j$ (Nilsson and Maluszynski, 1995, Definition 1.17). The *application* $X\sigma$ of a substitution $\sigma$ to a variable $X$ is defined as follows:

$$X\sigma := \begin{cases} t & \text{if } X/t \in \sigma, \\ X & \text{otherwise} \end{cases}$$

Let $\sigma$ be a substitution $\{X_1/t_1, \ldots, X_n/t_n\}$ and $E$ a term or a formula. According to (Nilsson and Maluszynski, 1995, Definition 1.18), the application $E\sigma$ of $\sigma$ to $E$ is the term/formula obtained by simultaneously replacing $t_i$ for every free occurrence of $X_i$ in $E\,(1 \leqslant i \leqslant n)$. $E\sigma$ is called an *instance* of $E$.

A *clause* is a formula (Nilsson and Maluszynski, 1995, Definition 2.1)

$$\forall(L_1 \vee \ldots \vee L_n)$$

where all $L_i$ are an atomic formula (positive literal) or the negation of an atomic formula (negative literal). A *definite clause* is defined as a clause that contains exactly one positive literal. Hence, a definite clause is a formula of the form (Nilsson and Maluszynski, 1995, Chapter 2.2):

$$\forall(A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n)$$

As a notational convention, a definite clause is written as follows:

$$A_0 \leftarrow A_1, \ldots, A_n\,(n \geqslant 0)$$

For such a definite clause, $A_0$ is called the *head* and $A_1, \ldots, A_n$ is called the *body* of the clause.

Furthermore, '$\top$' denotes an atomic formula that is true in every interpretation. A definite clause for which $n = 0$ is notated as $A_0 \leftarrow \top$. Moreover, '$\not\vdash$' is used in definite clauses and definite goals as the *negation as (final) failure* operator as introduced by Clark (1987) and implemented in several Prolog systems.

A *definite program* is defined as a finite set of definite clauses (Nilsson and Maluszynski, 1995, Definition 2.2). A definite program can be queried in terms of *definite goals*. A definite goal is denoted as follows:

$$\leftarrow A_1, \ldots, A_n$$

A definite goal can be seen as an existential question that is answered by a definite

program by means of trying to construct a logical consequence of the program which is an instance of a conjunction of all subgoals of the goal (i.e., an instance of $A_1 \wedge \ldots \wedge A_n$).

Moreover, the proposed domain model (i.e., the state and the activities model) is based on the assumption that two distinct constants necessarily refer to two distinct objects in the world. This assumption is usually called the *unique-name assumption*.

## 3.2. Outline of the Domain Model

A domain model represents all information that is used by the plan-based control architecture. It is, as already described, composed of a state and an activities model. Therefore, the domain model is defined as follows:

**Definition 3.1.** A domain model is a 2-tuple $D_M = (s_M(D_M), act(D_M))$ whereby

- $s_M(D_M)$ is the state model of $D_M$,
- and $act(D_M)$ is the activities model of $D_M$.

This definition serves as a coarse outline of the domain model until the state and activities model are described in more detail. The state model is described in this chapter and the activities model is described in Chapter 4.

## 3.3. Basic State Model

Several non-classical planning systems use axiomatic inference techniques to reason about the state of the world (Ghallab et al., 2004). Often the well investigated definite-clause inference techniques are used. Usually axiomatic inference is supported by calling a theorem prover as a subroutine of the overall planning process. The exploited knowledge representation and theorem proving systems (e.g., PDDL axioms (Thiébaux et al., 2005)) often rely on the *closed world assumption* (CWA). However, if we want to enable a planner to reason about unknown information in a partially known domain, then we need a state model and theorem proving system that are not based on the CWA. Particularly, we need an appropriate handling of negation.

As an alternative to implicitly representing negative information (e.g., by using the negation-as-failure semantics (Clark, 1987))—as often done by definite-clause theorem provers—it is possible to extend the syntax of definite clauses for the purpose of supporting the explicit representation of negative information. It has been stated in literature that this approach is often practically infeasible, because of the sheer magnitude of negative facts that would have to be stated (Subrahmanian, 1999). Nevertheless, this only holds under the assumptions that

1. a complete state model should be represented,

2. and it is not possible to define a complete representation for local parts of the overall model.

However, with respect to the context and objectives of this work neither of these two assumptions is fulfilled, since it is intended to develop an adequate domain model for incompletely known domains, and the proposed domain model—as introduced later—permits the explicit representation of complete parts at the level of predicates. Thus, it is reasonable to directly represent negative information in the context of this work.

The following two special kinds of terms are introduced in order to extend the syntax of definite clauses so that negative information can be explicitly represented: *literals* and *statements*. Furthermore, the following three special functors are added to the alphabet of the state model in order to define literals and statements: '`neg`', '$\wedge$' and '$\vee$'.

Let $\mathcal{A}_0$ be an alphabet that does not contain the functors '`neg`', '$\wedge$' and '$\vee$'. A term over the alphabet $\mathcal{A}_0$ (see (Nilsson and Maluszynski, 1995, Definition 1.1)) is called a *basic term*.

Based on that, a literal is defined as follows:

**Definition 3.2** (literal)**.** If $t$ is a basic term, then $t$ and `neg` $t$ are called a literal.

Moreover, a statement is defined by the following definition:

**Definition 3.3** (statement)**.** A term $st$ is called a *statement* iff it can be constructed by the following rules:

- $st$ is a literal
- $st = (\texttt{neg}\ st')$ and $st'$ is a statement
- $st = (st' \wedge st'')$ and $st'$ as well as $st''$ are statements
- $st = (st' \vee st'')$ and $st'$ as well as $st''$ are statements

In order to improve the readability, the functors '$\wedge$' and '$\vee$' are written in infix notation, and the functor '`neg`' is written in prefix notation. Thus, a statement $(st' \wedge st'')$ is defined as $\wedge(st', st'')$; a statement $(st' \vee st'')$ is defined as $\vee(st', st'')$; and a statement $(\texttt{neg}\ st')$ is defined as $\texttt{neg}(st')$.

Statements—including literals—are defined as special terms so that definite-clause reasoning technology can be used to derive instances of statements. Conceptually a literal essentially is what is known as a literal in first order logic. Similarly, a statement essentially is what is known as a first order logic sentence. Statements are always implicitly quantified. Statements in a definite clause are (implicitly) universally quantified, whereas statements in a definite goal are (implicitly) existentially quantified.

Similar to PDDL with PDDL axioms, the proposed state model enables domain experts to express factual (e.g., Bob's mug is in the kitchen) and axiomatic (e.g.,

Bob's mug is in room $X_1$ if Bob's mug is on table $X_2$ and $X_2$ is in room $X_1$) knowledge. Due to the objective to deal with open-ended domains, the domain model additionally supports the explicit representation of negative information. Moreover, it supports the flexible extension of the representation language of a state model by additional constructs. These additional constructs are intended to constitute higher level (conceptual) knowledge and are called *concept relations*. In principle, the state model can be extended to support any conceptual knowledge as long as one can compile this information to the underlying knowledge representation formalism, namely a definite program. This feature is exploited in the following part of this thesis by successively adding support for additional concept relations that are intended to deal with the special requirements of open-ended domains. For example, the explicit representation of subsumption-relations (e.g., A mug is an object) is supported by the state model in terms of an additional state model concept relation.

We introduce the special predicate symbol $d$ so that we can use definite clause reasoning to derive instances of statements. The idea is that an instance of a statement $st$ can be derived with respect to a state model $s_M$ if $d(st)$ can be derived from the definite program that is constituted by the proposed state model. A *state model* is formally defined as follows:

**Definition 3.4** (state model)**.** A *state model* is a quadruple $s_M = (F, C, R_D, R_G)$. Let $F_t$ be a set of ground literals and $C_t$ be a set of basic terms such that $F_t \cap C_t = \varnothing$. Then, $F$ is defined as $F = \{d(f) | f \in F_t\}$ and $C$ is defined as $C = \{d(c) | c \in C_t\}$. $R_D$ is a set of definite clauses $d(l) \leftarrow d(s)$ such that $l$ is a literal and $s$ is a statement. $R_G$ is a set of definite clauses. ${s_M}^{dp} = \{f \leftarrow \top | f \in F \cup C\} \cup R_D \cup R_G$ is the definite program constituted by the state model.

A state model $s_M$ is represented by the four sets $F$, $C$, $R_D$, and $R_G$. $F$ represents a set of facts about the state of a domain. $C$ contains additional conceptual knowledge. $R_D$ represents domain-specific rules (i.e., domain-specific axiomatic knowledge). In contrast, $R_G$ represents generic (i.e., domain-independent) rules (e.g., $(A \wedge B)$ holds if $A$ and $B$ hold). $F$, $C$, and $R_D$ are intended to be specified by a domain expert in order to model the state of a certain domain. $R_G$, however, represents generic rules that are defined together with the supported state model language constructs in order to be able to map these constructs to the level of definite clauses. A state model is illustrated in Figure 3.1. If it does not lead to ambiguities, then the special predicate symbol $d$ is omitted (e.g., like in Figure 3.1), and a definite clause $d(l) \leftarrow d(st)$ is written as $l \leftarrow st$.

The fact that a state model constitutes a definite program has the advantage that the process of deriving information from the state model can be reduced to the well-know process of deriving information from a definite program. From a more practical perspective, one can additionally benefit from the actuality that several highly optimized Prolog implementations are available that can automatically determine whether an instance of a definite goal is derivable or not. The derivability of a statement is defined as follows:

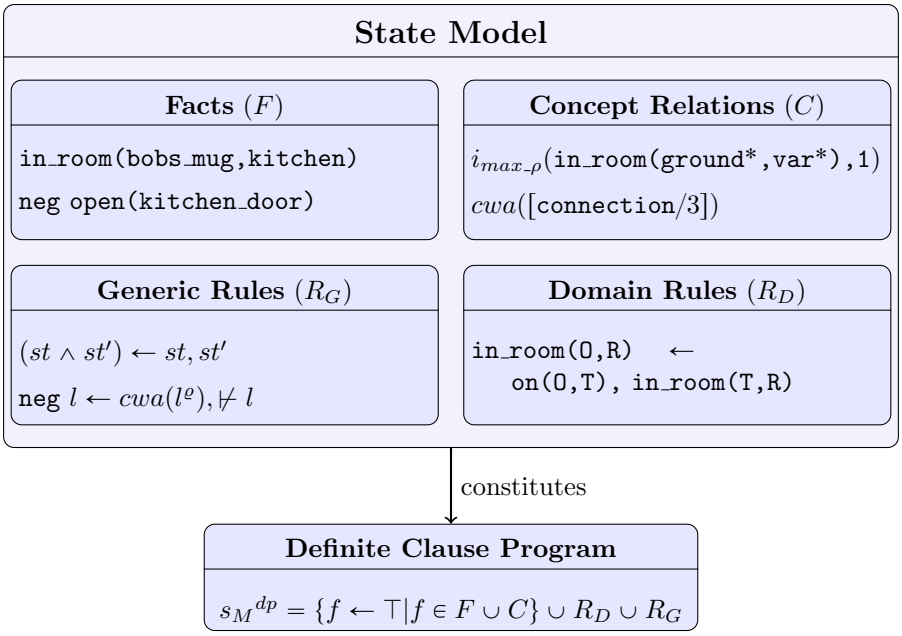



Figure 3.1.: Illustration of a state model.

**Definition 3.5** (derivable)**.** A statement $st$ is *derivable* with respect to a state model $s_M$ and a (grounding) substitution $\sigma$ (denoted as $s_M \vdash_\sigma st$) iff $d(st)\sigma$ is derivable with respect to ${s_M}^{dp}$.

The set of all instances of a statement $st$ that are derivable with respect to a state model $s_M$ is denoted as $\widetilde{\vdash}(s_M, st)$ respectively as $\widetilde{\vdash}(st)$ if the respective state model is apparent. More precisely, $\widetilde{\vdash}(s_M, st)$ is defined as follows:

$$\widetilde{\vdash}(s_M, st) := \{st\sigma \mid s_m \vdash_\sigma st\} \tag{3.1}$$

In order to specify the derivability of statements, the following generic rules are added to the set $R_G$ of a state model $s_M = (F, C, R_D, R_G)$:

$$d(st \wedge st') \leftarrow d(st), d(st') \tag{GR1}$$

$$d(st \vee st') \leftarrow d(st) \tag{GR2}$$

$$d(st \vee st') \leftarrow d(st') \tag{GR3}$$

$$d(\texttt{neg neg } st) \leftarrow d(st) \tag{GR4}$$

$$d(\texttt{neg } (st \wedge st')) \leftarrow d(\texttt{neg } st \vee \texttt{neg } st') \tag{GR5}$$

$$d(\texttt{neg } (st \vee st')) \leftarrow d(\texttt{neg } st \wedge \texttt{neg } st') \tag{GR6}$$

These rules determine what instances of a statement can derived. In particular,

the handling of the introduced negation operator '`neg`' is specified. Rule GR1 defines the handling of conjunction, whereas GR2 as well as GR3 define the handling of disjunction. The well-known double negation elimination rule (see for example Ben-Ari (2001)) is represented by GR4, and two of the *De Morgan rules* (Russell and Norvig, 2010, page 298) are defined by GR5 and GR6. Please note that it sometimes can be easier to understand generic rules if you read them as a definite program.

As already pointed out, a domain modeller has the opportunity to define domain-specific axioms. Axioms are known to be an important feature of domain languages (Thiébaux et al., 2005). Two example axioms are defined as follows:

$$\texttt{in\_room(O,R)} \leftarrow \texttt{on(O,T)},\ \texttt{in\_room(T,R)} \tag{DR1}$$

$$\texttt{neg in\_room(O,R)} \leftarrow \texttt{in\_room(O,R2)},\ \texttt{R2} \neq \texttt{R} \tag{DR2}$$

DR1 represents the fact that an object is in a room `R` if it is lying on a table which is in room `R`. DR2 is an example for the explicit representation of negative information. It represents the fact that an object can only be in one room at a given point in time.

## 3.4. Reasoning Process

The facts, domain specific rules, and concept relations of a state model are domain specific. They are usually specified by a domain expert or the result of a learning process. In contrast, the generic rules are domain independent. They can be seen as the program of the state model's *reasoner*. The reasoner that is constituted by the generic rules is called *ACogReason*. It is successively defined in this chapter by means of adding generic rules to the state model. ACogReason provides the basic reasoning services for the overall plan-based control architecture developed in the context of this work.

Figure 3.2 illustrates the idea of the reasoning process. The reasoner and the domain specific part of the state model together constitute a definite program. If the state model is queried in form of a definite goal, then this goal and the definite program constituted by the state model are the input of a definite-clause reasoner (e.g., a Prolog system). Building the reasoner on top of a definite-clause reasoner has the following advantages:

1. Definite-clause reasoning is based on mathematical logic, which has a well-understood, well-developed theory (Sterling and Shapiro, 1994).

2. The reasoning about the complete parts of the state model can be provided by the underlying definite-clause reasoner.

3. A multitude of highly optimized and mature Prolog systems are available that can act as a definite-clause reasoner. The performance of the proposed

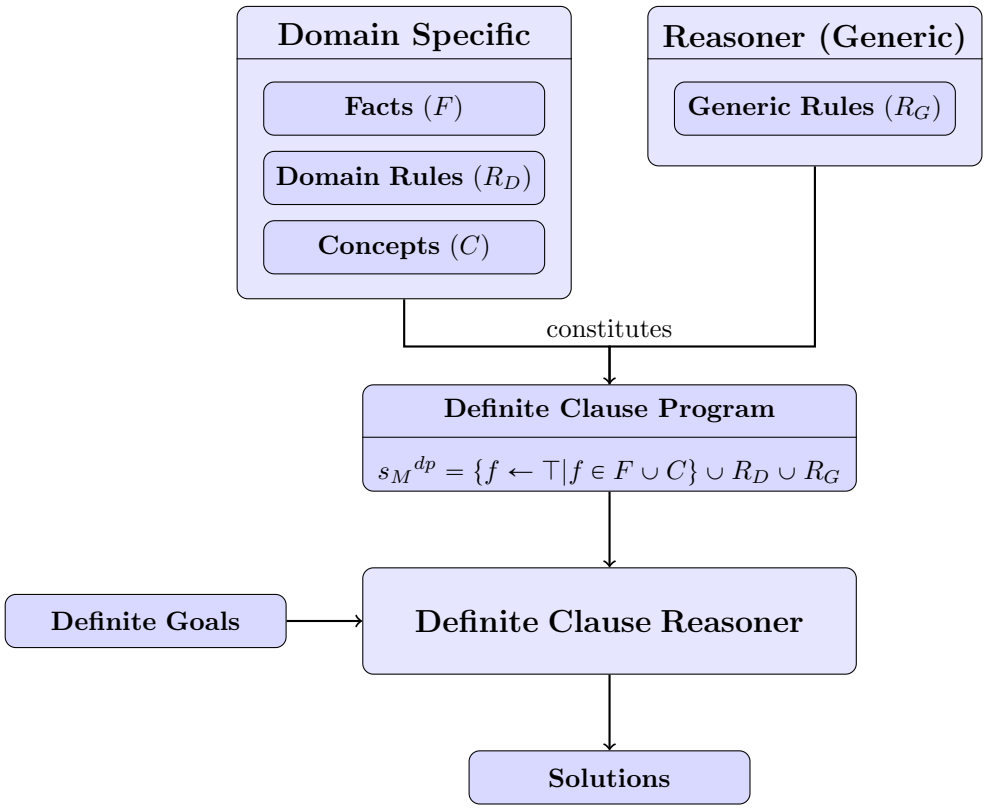



Figure 3.2.: Illustration of the reasoning process.

reasoning processes extensively benefits from the high performance of state-of-the-art Prolog systems.

## 3.5. Consistency

Nguyen (2008) distinguishes between two levels of inconsistency: *syntactic* and *semantic* inconsistency. Syntactic inconsistency refers to a set of contradictory formulas of a logic-based knowledge base. On the semantic level, formulas refer to a concrete slice of reality that is their interpretation. Semantic inconsistency occurs if a fact and its negation can be derived with respect to this slice of reality.

Due to the fact that the state model features the explicit representation of negative information, it is in principle possible to construct a syntactically inconsistent state model. This means that it is possible to construct a state model where a statement *st* and its negation `neg` *st* are derivable. However, semantic inconsistencies may also occur in CWA-based representations. For example, one can create a CWA-based state model such that `open(door1)` and `closed(door1)` are derivable. Thus, one also has to deal with inconsistencies in CWA-based models. Further-

more, note that the explicit representation of negation by means of the definite clause `neg open(door1)` ← `closed(door1)` has the advantage that one can detect the semantic inconsistency via syntactic techniques.

It is desirable to support domain modellers with software tools in order to prevent the creation of inconsistent state models, but automatically ensuring the syntactic consistency of the proposed state model is out of the scope of this work and not further addressed. Thus, ensuring consistency or dealing with inconsistencies needs to be addressed during the state model construction process.

The fact that a state model $s_M$ is consistent is denoted as $\mathfrak{c}(s_M)$. In the context of this work, it is implicitly always assumed that a state model is consistent.

## 3.6. Conceptualizing Open-Endedness

CWA-based knowledge representation and reasoning systems (e.g., Prolog) can in principle also be used in open-ended domains. Nevertheless, in open-ended domains one has to consider that it is possible that true instances of a statement "exist" but cannot be derived due to a lack of knowledge. CWA-based approaches are—by definition—unable to reason about unknown (i.e. non-derivable) but possibly true information. More precisely, it is unfeasible for CWA-based systems to distinguish between instances of statements that cannot be derived because the existence is impossible and instances of a statement that might be derivable if additional information about the state of the domain were available.

*Example 3.1* For example, let us assume that the following set of literals is derivable from the state model $s_M$ of an agent:

$$\{\texttt{mug}(\texttt{bobs\_mug}), \texttt{in\_room}(\texttt{bobs\_mug}, \texttt{kitchen})\}$$

If one would try to derive whether a true instance of

$$\texttt{mug}(\texttt{X}) \wedge \texttt{color}(\texttt{X}, \texttt{red})$$

exists with respect to $s_M$, then the only information a CWA-based reasoner can provide is that such an instance cannot be derived. Nevertheless, in principle there are two possible situations in which an instance of this statement exists. It might be

1. possible that Bob's mug is red, or
2. it might be possible that there is an additional (i.e., non-derivable) mug that is red.

For the purpose of also exemplifying the case where the existence of an instance is impossible, let us take a look at the following statement:

$$\texttt{in\_room}(\texttt{bobs\_mug}, \texttt{office})$$

Once again, the only thing a CWA-based reasoner can tell us about the literal is the fact that it is not derivable. However, in this case the existence of a true instance is impossible if one makes the reasonable assumption that Bob's mug cannot be in two different rooms at same point in time as specified by DR2.

Summing up, the CWA leads to a strong limitation that makes it hard to reason about unknown information. The objective of the proposed open-ended state model is to enable the distinction between situations in which the existence of a non-derivable instance of a statement is impossible and situations in which additional information might make non-derivable instances derivable. If we want to enable such a reasoning, then we need an open-ended state model. This section introduces a new open-ended state model that is based on the following three novel concepts: an *F-extension*; an *open-ended literal*; and a *possibly-derivable statement*.

For the purpose of reasoning about open-ended domains, one has to reason about possible extensions of a state model. In this connection, only extensions are considered that are constituted by adding factual knowledge (i.e., a set of literals) to a state model. These extensions are called *F-extensions* and are formally conceptualized as follows:

**Definition 3.6** (F-extension)**.** A state model $s'_M = (F', C, R_D, R_G)$ is called an *F-extension* of $s_M = (F, C, R_D, R_G)$ (denoted as $s_M \sqsubseteq_F s'_M$) iff $F \subseteq F'$ and $\mathfrak{c}(s_M) \Rightarrow \mathfrak{c}(s'_M)$.

In other words, one can create an F-extension of a state model by adding literals such that a consistent state model stays consistent. Furthermore, literals for which the existence of non-derivable instances is possible are called *open-ended*:

**Definition 3.7** (open-ended literal)**.** A literal $l$ is called *open-ended* with respect to a state model $s_M$ (denoted as $l^{\sqsubseteq}$) iff there is an instance $l\sigma$ of $l$ and a state model $s'_M$ such that $s_M \sqsubseteq_F s'_M$ and $l\sigma \in (\widetilde{\vdash}(s'_M, l) \setminus \widetilde{\vdash}(s_M, l))$.

Please note that for a ground literal the following holds:

**Remark 3.1.** If $l$ is ground, then $l$ is open-ended iff neither an instance of $l$ nor of `neg` $l$ is derivable.

Let us recall the situation of Example 3.1 in order to exemplify the concept of an open-ended literal. The literals `mug(X)` and `color(X,red)` are examples of open-ended literals, because the existence of non-derivable mugs and red things is possible. In contrast, `mug(bobs_mug)` is not open-ended, since the only possible instance is already derivable.

Let *ground*($l$) be a meta-predicate that holds iff $l$ is ground and *non-ground*($l$) be a meta-predicate that holds iff $l$ is non-ground. The following two clauses constitute a first attempt to specify an open-ended literal by means of a set of definite-clauses:

$$d(l^{\sqsubseteq}) \leftarrow \textit{non-ground}(l) \tag{GR7}$$

$$d(l^{\sqsubseteq}) \leftarrow \textit{ground}(l), \not\models d(l), \not\models d(\texttt{neg}\ l) \tag{GR8}$$

In other words, a literal $l$ is open-ended if it is non-ground (GR7); or if it is ground and neither $l$ nor $\texttt{neg}\ l$ can be derived (GR8). Domain specific information can be used to determine that not every non-ground literal is open-ended. Nevertheless, additional domain model constructs are necessary to integrate these kind of domain specific information. These additional constructs will be introduced later in Section 3.9. For the current understanding, Rule GR7 is sufficient and will be revised in Section 3.9.

Based on the definition of an open-ended literal, a *possibly-derivable statement* is defined as follows:

**Definition 3.8** (possibly-derivable statement)**.** A statement $st$ is possibly-derivable with respect to a state model $s_M$, a substitution $\sigma$, and a set of open-ended literals $L_x$ (denoted as $\Diamond(st, L_x)$) iff

- $L_x$ is empty, and $st$ is derivable with respect to $s_M$ and $\sigma$;
- or $L_x$ is nonempty, and if a new instance $(l\sigma)\sigma'$ is derivable for all $l \in L_x$ with respect to an F-extension $s'_M$ of $s_M$ and the empty substitution $\varnothing$, then a new instance $(st\sigma)\sigma'$ of $st$ is derivable with respect to $s'_M$ and the empty substitution $\varnothing$.

The symbol '$\Diamond$' is—like $d$—a special predicate symbol. The idea is that a possibly-derivable instance of a statement $st$ can be derived with respect to $s_M$ and $L_x$ if $\Diamond(st, L_x)$ can be derived with respect to the definite program constituted by $s_M$. From the first part of this definition one can directly infer the following:

**Remark 3.2.** If a statement $st$ is derivable with respect to a state model $s_M$ and a substitution $\sigma$, then it is possibly-derivable with respect to $s_M$, $\sigma$, and the empty set (of open-ended literals).

A possibly-derivable statement constitutes the partition of a logical statement into a derivable and an open-ended part (i.e., a set of open-ended literals). This partition determines what additional information is necessary in order to derive an additional (i.e., non-derivable with respect to the state model at hand) instance of a given statement. Note that there may be more than one way to partition a statement into a derivable and an open-ended part.

Let us assume that we have the same state model $s_M$ as introduced in Example 3.1 and would like to know whether the statement $st = \texttt{mug(X)} \wedge \texttt{color(X, red)}$ is possibly-derivable. Thus, we are looking for a red mug. In this example, there are two different situations in which $st$ is possibly-derivable. In the first situation, $X$ is substituted with `bobs_mug`, and $st$ is possibly-derivable with respect to $s_M$ and the resulting set of open-ended literals $\{\texttt{color(bobs\_mug, red)}\}$. In the second

situation, the fact that there might exist an unknown red mug can be exploited, and $st$ is possibly-derivable with respect to $s_M$ and the set of open-ended literals $\{\texttt{mug(X)}, \texttt{color(X, red)}\}$.

Let $literal(l)$ be a meta-predicate that holds iff $l$ is a literal. The following generic rules are introduced in order to be able to derive possibly-derivable statements:

$$\Diamond(st, L_x) \leftarrow \Diamond(st, \varnothing, L_x) \tag{GR9}$$

$$\Diamond(st, L_x, L_x) \leftarrow literal(st), d(st), \forall_{l \in L_x} : d(l^{\sqsubseteq}) \tag{GR10}$$

$$\Diamond(st, L_x, L_x \cup \{st\}) \leftarrow literal(st), d(st^{\sqsubseteq}) \tag{GR11}$$

$$\Diamond((st \wedge st'), L_x, {L_x}') \leftarrow \Diamond(st, L_x, {L_x}''),\ \Diamond(st', {L_x}'', {L_x}') \tag{GR12}$$

$$\Diamond((st \vee st'), L_x, {L_x}') \leftarrow \Diamond(st, L_x, {L_x}') \tag{GR13}$$

$$\Diamond((st \vee st'), L_x, {L_x}') \leftarrow \Diamond(st', L_x, {L_x}') \tag{GR14}$$

$$\Diamond(\texttt{neg}\ (st \wedge st'), L_x, {L_x}') \leftarrow \Diamond((\texttt{neg}\ st \vee \texttt{neg}\ st'), L_x, {L_x}') \tag{GR15}$$

$$\Diamond(\texttt{neg}\ (st \vee st'), L_x, {L_x}') \leftarrow \Diamond((\texttt{neg}\ st \wedge \texttt{neg}\ st'), L_x, {L_x}') \tag{GR16}$$

GR10 and GR11 specify under what conditions a literal is possibly-derivable. The general idea is that a literal is possibly-derivable if it is derivable or open-ended. Thus, every open-ended literal is possibly-derivable, because for every open-ended literal it is possible that there is a consistent extension of the current domain model so that it is derivable with respect to this extension. Note that a (non-ground) literal can be both derivable and open-ended. The second argument of the ternary '$\Diamond$' predicate[1] denotes the set of open-ended literals of the previous part of a statement and initially is empty (see GR9). Including this argument into the recursive definition is necessary in order to consider the possible dependencies between different parts of a statement. To be more precise, it has to be ensured that all literals that have been "chosen" to be in the open-ended part of a statement stay open-ended after additional substitutions. This is exactly what is done in GR10 by means of ensuring that substitutions that are necessary in order to derive an instance of $st$ do not affect the open-endedness of the literals in $L_x$. Besides the correct handling of the set of open-ended literals, GR12 - GR16 essentially describe well-known rules of first order logic.

[1] In the Prolog community, such an argument is typically called an *accumulator* (Sterling and Shapiro, 1994).

## 3.7. Additional Meta-Predicates

The proposed state model supports a set of meta-predicates that have a special semantics. The meta-predicates '$\top$', '$\bot$', '$\not\models$', '*ground*$(X)$', '*non-ground(X)*' and their semantics have already been defined in Section 3.1 and 3.6. Meta-predicates can be used as a prefix (e.g., '$\not\models$') or an infix operator. Additional meta-predicates featured by ACogDM are introduced in this section.

### 3.7.1. Only Consider One Solution

In some situations, one might be only interested in finding any instance of a given statement. The '**once**' meta-predicate can be used to represent such a situation. For a statement $st$, the meta-predicate can be used in prefix notation as follows:

**once** $st$

If the given statement is derivable, then the '**once**' meta-predicate is handled like in Prolog (Deransart et al., 1996): The reasoner returns the first derived instance and ignores (possibly existing) additional solutions. However, if no instance of the statement is derivable, then the meta-predicate has no effect, and the reasoner returns all existing possibly-derivable instances of a statement.

Only considering one instance of a precondition can significantly improve the performance of the planning process. However, this comes at the cost of possibly ruling out relevant planning alternatives. Hence, the usage of the '**once**' predicate can prevent the planner from finding solutions and should be used carefully.

### 3.7.2. Calling External Components

In order to perform a task in real-world situations, an agent often needs to solve subproblems that differ in terms of the required reasoning techniques. For example, for a typical pick-and-place service robotic task, a robot needs to solve path and motion planning as subproblems. These problems are known to have a complex combinatorial structure, and are hard to solve for heuristic-search-based planners (Helmert and Röger, 2008). The problem of integrating path and motion planning into an overall task planning process is usually tackled in a top-down manner so that general purpose task planning, and dedicated path and motion planning run in isolation. In this way, task planning is simplified by ignoring low-level details. Unfortunately, this approach has the disadvantage that resulting plans on the task-planning layer may be inefficient or even infeasible due to unconsidered low-level constraints. For example, consider a situation in which a robot wants to put a mug of coffee down and pick an empty plate up from the same table. If the kinematic constraints of the robot are not considered by a task planner, then the task planner is unaware of the fact that placing the object in a certain way (e.g., at a certain position) effects the difficulty—or even the feasibility—of the subsequent pick-up

task. For example, placing the mug directly in front of the empty plate can make it more difficult—or even impossible—to subsequently pick the plate up.

The opposite strategy is to precompute all information that is possibly relevant for a task planner in a bottom-up manner by low-level reasoners. Nevertheless, this approach is often practically impossible due to the sheer magnitude of information that would have to be precomputed.

A third alternative is to invoke external reasoners only when necessary, thereby enabling the planner to integrate relevant information from dedicated reasoners into the task planning process. However, this leads to the following question: When is it reasonable to invoke a certain external reasoner? The simplest way to deal with this question is to encode the call of external components into the domain description. The '**call**' meta-predicate is featured by ACogDM in order to support this approach. Every literal of the form

$$\textbf{call}\ l$$

is evaluated by calling $l$. For example, the following literal is evaluated by means of calling an external path planner for the purpose of determining how expensive it is to move from a position $Pos1$ to a goal position $Pos2$:

$$\textbf{call}\ path\_planner :: (Pos1, Pos2, Cost)$$

Encoding the calling of external reasoners into the domain description is not a new idea. It is also supported by JShop2 (Ilghami, 2006) as well as by a PDDL extension (Dornhege et al., 2009).

However, predefining the calling of external components in the domain model has the disadvantage that it massively increases the size of the model. Explicitly defining when to acquire what information from external components usually is practically impossible, since the state spaces are too large for realistic domains. Thus, autonomous agents need to be able to autonomously decide when to acquire what information from which external source. How this can achieved is described in Section 5 and 6.

### 3.7.3. Evaluating Arithmetic Expressions

The following ISO Prolog meta-predicates for arithmetic comparison are supported by the state model in infix notation:

- `'=:='` (equal)
- `'=\='` (not equal)
- `'<'` (less than)
- `'=<'` (less than or equal to)

- `'>'` (greater than)
- `'>='` (greater than or equal to)

Moreover, the meta-predicate '**is**' can be used to evaluate arithmetic expressions like in Prolog (Deransart et al., 1996). In this work, arithmetic expressions are evaluated by the underlying Prolog system (SWI-Prolog).

### 3.7.4. Not Unifiable

The fact that two terms $t, t'$ have no *unifier* (Nilsson and Maluszynski, 1995, Definition 3.1) can be expressed by using the meta-literal '$\neq$' in the following way:

$$t \neq t'$$

The statement $t \neq t'$ is derivable iff $t$ and $t'$ are not unifiable.

### 3.7.5. List Operations

Furthermore, the fact that a term $t$ is not unifiable with a member of the list $[t_1, \ldots, t_n]$ is defined by statements of the following form:

$$t \textbf{ notin } [t_1, \ldots, t_n]$$

### 3.7.6. Accessing the Domain Model

Access to the underlying domain model $M$ is provides by the following meta-predicate:

$$\textbf{dm } M$$

The variable $M$ is instantiated with the underlying domain model. Providing access to the whole domain model is, for example, useful in order to forward the domain model to an external reasoning component. A more concrete example for the usage of the '**dm**' meta-predicate can be found in Chapter 7.

### 3.7.7. Derivability Status

The fact that a literal $l$ is open-ended can be stated in the following way with the '**oe**' meta-predicate:

$$\textbf{oe } l$$

In some situations, it can be reasonable to only consider derivable instances of a statement. The fact that the reasoner should only consider derivable instances of

a statement $s$ (i.e., instances that are only possibly-derivable are ignored) can be expressed with the '**deriv**' meta-predicate as follows:

**deriv** $s$

Furthermore, the fact that a statement $s$ possibly holds can be expressed in the following way:

**possibly** $s$

A statement '**possibly** $s$' holds iff $s$ is possibly-derivable. Thus, for a statement of the form '**possibly** $s$' ACogReason only checks whether $s$ is possibly-derivable and does not consider knowledge acquisition tasks for the corresponding set of open-ended literals.

## 3.8. Interpretation Model

How a literal is handled by the reasoner depends on its *interpretation model*. The interpretation model of a literal can be *open-world assumption* (*owa*), *closed world assumption* (*cwa*), or *reasoning*. In the following, the interpretation model of a literal $l$ with respect to a domain model $D_M$ is denoted as $i_M(l, D_M)$. If the domain model is apparent or not important for a certain consideration, then it is omitted, and the interpretation model of a literal $l$ is denoted as $i_M(l)$. By default, all predicates are interpreted based on the interpretation model *owa*. This means that the reasoner by default does not assume that all possible instances of a given literal can be derived with respect to a state model.

Nevertheless, in order to combine the best of both worlds, it is possible to define on the predicate level if a literal should be interpreted based on the CWA, or the OWA. For example, imagine a predicate `connection(R1,D,R2)`, which describes that room `R1` is connected via door `D` with room `R2`. The relation that is represented by this predicate is rather static, thus even in dynamic unstructured environments it is possible to equip an artificial agent a priori with all true ground instances of this relation. In this case, it is reasonable to define the interpretation model of the `connection` predicate as *cwa*. This definition implies that `neg connection`$(\texttt{R1}, \texttt{D}, \texttt{R2})$ holds iff `connection(R1,D,R2)` cannot be derived—which in fact is the negation-as-failure semantics as introduced by Clark (1987). Predicate-based (i.e., local) closed world assumptions reduce the lack of knowledge and can significantly improve the performance of the plan generation and knowledge acquisition process.

Exploiting complete local parts of a logical theory is not a completely new idea, because it has already been proposed by Etzioni et al. (1997) and is also featured by PowerLoom (Chalupsky et al., 2010).

A predicate is symbolically represented as $[name/n]$, where *name* is the name of the predicate, and $n$ denotes the arity. The predicate of a literal $l$ is denoted as $l^{\varrho}$. The fact that a predicate is interpreted with respect to the CWA is represented by terms of the form $cwa([name/n])$. Thus, all predicates that are not defined as

being interpreted with respect to the CWA are—by default—interpreted based on the OWA.

Literals are interpreted based on the OWA in order to enable the reasoner to reason about extensions of a domain model. However, for all meta-predicates—except the **once** predicate—it is not reasonable to do that, since

- their interpretation is independent of the domain model at all (e.g., this holds for '$\neq$', '$\top$', '$\bot$', '*ground*$(X)$', '*non-ground(X)*');
- or they should only be interpreted according to the current domain model (e.g., this holds for '**dm** $M$', '**oe** $l$', '**deriv** $s$', '**possibly** $s$');
- or it is unclear whether and how the result of the reasoning process depends on the domain model (e.g., this holds for the '**call**' meta-predicate).

In other words, the interpretation of these meta-predicates is only possible by means of a dedicated reasoning process. In contrast to other literals, it is not possible to consider extensions of the state model at hand for the purpose of determining additional instances of the mentioned meta-predicates. The interpretation model for these literals is called *reasoning*. For a ground literal with the interpretation model *reasoning*, it is assumed that the corresponding reasoning process can always decide whether the literal holds with respect to a given domain model. However, it is not assumed that a corresponding reasoner can derive all possible instances of a non-ground literal with the interpretation model *reasoning*. Another way to explain the assumption made about the reasoning process that derives instances of a literal with the interpretation model *reasoning* is to assume that the reasoning process uses the *generate-and-test* paradigm (Sterling and Shapiro, 1994). In logic programming, the generate-and-test technique can be easily implemented by means of a conjunction of two literals in which one acts as the generator, and the other tests whether the generated instances hold, as in the following clause (Sterling and Shapiro, 1994, page 250):

$$find(X) \leftarrow generate(X), test(X)$$

In terms of the generate-and-test paradigm, it is assumed that the $test(X)$ procedure of a reasoner is complete. Nevertheless, the generation process denoted by $generate(X)$ may be incomplete. Usually the generation process is constituted by the evaluation of dependent literals in the context of a compound statement. For example, consider the following statement:

$$\texttt{mug(X)} \wedge \texttt{status(X,Y)} \wedge \texttt{Y} \neq \texttt{dirty}$$

In this example, the first two literals are first instantiated, since statements are evaluated from left to right. Thus, the third literal ('$\texttt{Y} \neq \texttt{dirty}$') is usually only evaluated after `Y` is instantiated. Backtracking can be used in order to possibly generate multiple instances of `Y`. In this way, the evaluation of the previous part of

the statement constitutes the generation process in terms of the generate-and-test paradigm.

Algorithm 3.1 specifies how the interpretation model of a literal $l$ is determined. The input of the algorithm is a literal and a domain model. The interpretation model of a literal '**once** $l'$' is the interpretation model of $l'$. Hence, the algorithm recursively calls itself for purpose of determining the interpretation model of $l'$ (lines 1-2). As already pointed out, the interpretation model for all other meta-predicates is *reasoning* (lines 3-4). For all other literals, the interpretation model is *cwa* if this is explicitly defined in the domain model (lines 5-6). Negative literals '*neg l*' need a special handling if the interpretation model $i_M(l)$ is *cwa* or *reasoning*. In this situation, '*neg l*' is defined as *reasoning* (7-8), since the interpretation is only possible by means of a corresponding reasoning process. The negation of a literal that has the interpretation model *cwa* is not also defined as *cwa*, since this would imply that the reasoner must be able to derive all instances of the negation. For example, if we consider that a robot agent knows all connections between the rooms of a building. Thus, the interpretation model of the corresponding predicate `connection(R1,D,R2)` is *cwa* and the robot can derive all valid instances of it. Moreover, it is possible to decide for any ground instance of `connection(R1,D,R2)` whether the negation (i.e., `neg connection(R1,D,R2)`) holds; since if `neg connection(R1,D,R2)`$\sigma$ is ground, then it is derivable iff `connection(R1,D,R2)`$\sigma$ is not derivable. This relation follows directly from the definition of the closed-world assumption. Nevertheless, for a non-ground negative literal `neg connection(R1,D,R2)`, it is not always clear how instances can be generated by the reasoner. Generating all such instances is particularly difficult in an open-ended domain, because there can be an infinite number of valid instances. Thus, the interpretation model of such a literal is by default defined as *reasoning* so that the reasoner does not have to be able to derive all instances of it. If none of the aforementioned conditions is fulfilled, then the interpretation model of a literal is *owa* (lines 9-10).

As already pointed out, the proposed state model supports the definition of closed-world assumptions at the level of predicates. Based on the notion of an interpretation model, the following generic rule is added to the state model specification so that predicate-based closed-world assumptions are properly handled by the state model:

$$d(\texttt{neg}\ l) \leftarrow d(i_M(l) = cwa), \nvdash d(l) \qquad \text{(GR17)}$$

In other words, the generic rule GR17 defines that the negation of a literal $l$ holds if the interpretation model of the literal is *cwa* and $l$ is not derivable. The same statement can be made for ground literals with the interpretation model *reasoning*. Therefore, the following rule is also added to the state model:

$$d(\texttt{neg}\ l) \leftarrow ground(l), d(i_M(l) = reasoning), \nvdash d(l) \qquad \text{(GR18)}$$

This rule defines that the negation of a literal holds if it is ground, has the interpretation model *reasoning*, and cannot be derived.

**Algorithm 3.1:** interpretation-model($l, D_M$)

**input**: domain model $D_M$, literal $l$
**Result**: interpretation model $i_M(l, D_M) \in \{owa, cwa, reasoning\}$

```
1  if l = once l' then
2      return interpretation-model(l', D_M);
3  else if l is a meta-predicate then
4      return reasoning;
5  else if cwa(l^g) is derivable with respect to s_M(D_M) then
6      return cwa;
7  else if l = neg l' AND interpretation-model(l', D_M)
     ∈ {cwa, reasoning} then
8      return reasoning;
9  else
10     return owa;
```

## 3.9. Additional Conceptual Knowledge

Statements are not just syntactical constructs. They constitute *concepts*. Supporting the representation on a conceptual meta-level—in contrast to representing knowledge on the level of definite clauses—has the advantage that it eases the knowledge engineering process, since domain experts can represent knowledge on a higher abstraction level that is often closer to the way they think about the domain. Defining that a certain predicate should be interpreted based on the closed-world assumption, as described in Section 3.8, is an example for knowledge representation on a conceptual meta-level.

In this section, the proposed state model is extended by additional concept relations that, so to speak, make it possible to reduce the open-endedness of the state model. The general idea is that one can reduce the open-endedness by means of exploiting additional domain knowledge so that the number of open-ended literals can be reduced. In this way, an agent improves its performance via ruling out (additional) impossible planning alternatives. For example, if an agent knows that `bobs_mug` is a mug and a mug cannot be a plate, then it does not have to consider state model extensions in which `bobs_mug` is a plate. Generally speaking, in this example, the reasoner exploits the fact that mugs and plates constitute two disjoint concepts.

Please keep in mind that the whole domain model is based on the unique-name assumption. Thus, two distinct constants necessarily designate two distinct objects in the world.

### 3.9.1. Maximum Number of Instances

With the current state model constituted by GR1 - GR18, every non-ground literal is open-ended (see GR7). To put it another way, it is assumed that an agent never knows all instances of a non-ground literal. However, this might not always be the case. On the conceptual level, domain constraints can limit the number of possible instances of a statement. For example, let us assume that the literal `in_room(bobs_mug,office)` is derivable. In this case, the non-ground literal `in_room(bobs_mug,X)` is not open-ended (i.e., no additional instance is possible) if we assume that an object can only be in one room at a given point in time. In order to be able to express these kinds of constraints, the language of the state model is extended by constructs of the form $i_{max_D}(l, n, c)$ such that $l$ is a literal, $n \in \mathbb{N} \cup \{\infty\}$, and $c$ is a statement. The term $i_{max_D}(l, n, c)$ explicitly represents that the literal $l$ can maximally have $n$ ground instances if $c$ holds. The following rules are added in order to "ground" this additional construct to the level of definite clauses:

$$d(i_{max_D}(l, n)) \leftarrow d(i_{max_D}(l, n, c)), d(c) \tag{GR19}$$

$$d(i_{max_D}(l, \infty)) \leftarrow \mathit{non\text{-}ground}(l), \nvdash d(i_{max_D}(l, n, X_1)) \tag{GR20}$$

$$d(i_{max_D}(l, 1)) \leftarrow \mathit{ground}(l) \tag{GR21}$$

The generic rule GR19 defines that the upper bound of the number of instances for a literal $l$ is $n$ if $i_{max_D}(l, n, c)$ as well as $c$ are derivable with respect to the same substitution. If the maximum number of instances cannot be derived, then it is unbounded for non-ground literals (see GR20) and one for ground literals (see GR21).

The explicit definition of the maximum number of instances is optional and not reasonable for all literals. For example, the maximum number of instances does not have to be explicitly defined for a literal that has the interpretation model *cwa*, since for a literal with the interpretation model *cwa*, the maximum number of instances always equals the number of derivable instances. The actual number of the maximum instances of a literal is derived from the state model. It is denoted by $i_{max}(l, n)$ for a literal $l$ where $n \in \mathbb{N} \cup \{\infty\}$.

If the maximum number can be derived based on the explicit definition of terms of the from $i_{max_D}(l, n, c)$, then the (actual) maximum number of instances equals $n$. This relationship is represented by the following generic rule:

$$d(i_{max}(l, n)) \leftarrow d(i_{max_D}(l, n)) \tag{GR22}$$

For a literal $l$ that is interpreted based on the CWA, the maximum number of instances equals the number of derivable instances. Hence, the following rule is added to the domain model:

$$d(i_{max}(l, |\widetilde{\vdash}(l)|)) \leftarrow d(i_M(l) = cwa) \tag{GR23}$$

The same assumption is also made for ground literals with the interpretation model *reasoning*. Due to the fact that a ground literal cannot have more than one instance, determining all instances of a ground literal is reduced to determining whether the literal holds with respect to the domain model at hand. In contrast to non-ground literals, it is not necessary to possibly generate additional instances. Therefore, it is assumed that the reasoner can derive all valid instances of a ground literal that has the interpretation model *reasoning*. This is represented by the following rule:

$$d(i_{max}(l, |\widetilde{\vdash}(l)|)) \leftarrow ground(l), d(i_M(l) = reasoning) \tag{GR24}$$

Based on the reasoning about the maximum number of possible instances of literals, the generic rule (GR7) can be replaced by the following, improved rule:

$$d(l^{\sqsubseteq}) \leftarrow non\text{-}ground(l), d(i_{max}(l, n)), d(|\widetilde{\vdash}(l)| < n) \tag{GR25}$$

In other words, a non-ground literal is open-ended if the number of derivable instances is less than the number of maximum instances.

How many instances of a literal can be derived often depends on the occurrence—or absence–of variables. For instance, consider the literal `in_room(O,R)` that defines that an object `O` is in a room `R`. If the first argument of the literal is a ground term (e.g., `bobs_mug`), then we know—based on our domain knowledge—that a resulting literal (e.g., `in_room(bobs_mug,R)`) can only have one instance, since an object cannot be in two different rooms at once. In this respect, the concrete value of the ground term is of no consequence. The number of possible instances of a literal `in_room(O,R)` is bounded by one whenever `O` is a ground term. In other words, the `in_room(O,R)` relation constitutes a function that maps an object to the room in which it is located. However, it is not necessarily an injective function. Thus, we cannot make statements about the maximum number of instances if the second argument is ground.

For the purpose of enabling it to restrict the possible number of instances of a literal based on the pattern of how variables and ground terms occur, the notion of an *instantiation scheme* is introduced. An instantiation scheme is defined as follows:

**Definition 3.9** (instantiation scheme)**.** An *instantiation scheme* is a ground literal that can contain the special constants *var** and *ground** whereby *var** is a place holder for any variable and *ground** is a place holder for any ground term.

The special constants *var** and *ground** enable it to define instantiation schemes that abstract from concrete variables and ground terms. If an instantiation scheme $\varphi$ can be viewed as an abstraction of a literal $l$, then $l$ is said to be *instantiable* with respect to $\varphi$. More precisely, an instantiable literal is defined as follows:

**Definition 3.10** (instantiable)**.** A literal or term $g$ is called *instantiable* with respect to an instantiation scheme $\varphi$ (denoted as $g \preceq_I \varphi$) iff one can construct a literal or

term $g'$ from $\varphi$ via replacing each $var^*$ by a new variable and each $ground^*$ by a ground term so that $g$ is an instance of $g'$.

For example, the literals `in_room(bobs_mug,R)` and `in_room(bobs_mug,kitchen)` are instantiable with respect to the instantiation schemes `in_room(`$ground^*$`,`$var^*$`)` and `in_room(bobs_mug,`$var^*$`)`. The literal `in_room(0,lab)`, however, is neither instantiable with respect to the scheme `in_room(`$ground^*$`,`$var^*$`)` nor with respect to `in_room(bobs_mug,`$var^*$`)`.

Instantiation schemes can be used to define an upper bound of the number of instances for a set of literals. The idea is that this upper bound is an upper bound for all literals that are instantiable with respect to the instantiation scheme. This idea leads to the following definition:

**Definition 3.11** (maximum instances (scheme))**.** Let $\varphi$ be an instantiation scheme and $n \in \mathbb{N} \cup \{\infty\}$. A literal of the form $i_{max\_\rho}(\varphi, n)$ defines that $n$ is the upper bound of the number of instances for all literals that are instantiable with respect to $\varphi$.

The following generic rule is added to the state model specification for the purpose of supporting constructs of the form $i_{max\_\rho}(\varphi, n)$:

$$d(i_{max_D}(l, n)) \leftarrow d(i_{max\_\rho}(\varphi, n)),\ l \preceq_I \varphi \tag{GR26}$$

For example, the fact that an object can only be in one room at a given point in time can be represented by the term $i_{max\_\rho}$`(in_room(`$ground^*, var^*$`),1)`. However, now we have a semantically redundant representation, because the conceptually same actuality is already specified by the domain specific rule DR2. Note that both representations have been introduced for different technical reasons. DR2 solely makes it possible to derive that all statements of the form '`neg in_room(obj,r)`' are true if it is known that `in_room(obj,r')` and `r'` $\neq$ `r` hold. In contrast, the definition of $i_{max\_\rho}$`(in_room(`$ground^*$`,`$var^*$`),1)` solely makes it possible to deduce that all statements with the instantiation scheme `in_room(`$ground^*, var^*$`)` can only have one instance. One can omit redundancies introduced by $i_{max\_\rho}$ via adding generic rules. For the purpose of achieving this, a few conceptualizations need to be introduced. First, the *lift* of a literal or term with respect to an instantiation scheme is defined as follows:

**Definition 3.12** (lift)**.** Let $\varphi$ be an instantiation scheme, $g$ be a literal or term that is instantiable with respect to $\varphi$, and $X^\star$ denote a new (i.e., unused) variable. The *lift* $\phi_\uparrow(g, \varphi)$ of $g$ with respect to $\varphi$ is defined as follows:

- $\phi_\uparrow(g, \varphi) := g$; if $\varphi = g$, or $g$ is ground and $\varphi = ground^*$.
- $\phi_\uparrow(g, \varphi) := X^\star$; if $\varphi = var^*$
- $\phi_\uparrow(g, \varphi) := f(\phi_\uparrow(u_1, u'_1), \ldots, \phi_\uparrow(u_m, u'_m))$; if $g = f(u_1, \ldots, u_m)$ and $\varphi = f(u'_1, \ldots, u'_m)$

Lifting a literal or a term $g$ with respect to an instantiation scheme $\varphi$ essentially means to construct a literal or term $g'$ such that $g'$ is instantiable with respect to $\varphi$ and $g$ is an instance of $g'$. Moreover, there is no literal or term $g'' \neq g'$ that is instantiable with respect to $\varphi$ such that $g'$ is an instance of $g''$. In other words, $g'$ is the most general literal or term that is instantiable with respect to $\varphi$ such that $g$ is an instance of it.

For example, lifting `in_room(bobs_mug,office)` with respect to the instantiation scheme `in_room(`*ground**`,`*var**`)` results in `in_room(bobs_mug,X)`.

According to Definition 3.12, lifting a term is achieved via recursively replacing terms by a new variable if the corresponding term of the instantiation scheme is *var**. The occurrence of duplicate variables imposes additional constraints on the possible instances of a literal. Thus, variables are replaced by new variables in order to ensure that the resulting literal contains no duplicate variables.

A literal is defined to be open-ended (see Definition 3.7) iff it is possible that an additional instance can be derived. In other words, a literal is known to be not open-ended if one can determine that the existence of such an instance is impossible. If $n$ instances of a literal that is instantiable with respect to $\varphi$ can already be derived, then the information encoded in constructs of the form $i_{max_\rho}(\varphi, n)$ can be used in order to prove that for all related literals, the existence of an additional instance is impossible.

More precisely, we can make the following proposition:

**Proposition 3.1.** *Let $l$ be a literal that is instantiable with respect to an instantiation scheme $\varphi$ and $i_{max\_\rho}(\varphi, n)$ be part of the state model $s_M$. $l$ is not open-ended with respect to $s_M$ if $|\widetilde{\vdash}(s_M, \phi_\uparrow(l, \varphi))| = n$.*

*Proof.* $l$ is by definition an instance of $\phi_\uparrow(l, \varphi)$. Thus, there is a substitution $\sigma$ such that $l = \phi_\uparrow(l, \varphi)\sigma$. If an additional instance $l\sigma'$ of $l$ is derivable, then an additional instance $\phi_\uparrow(l, \varphi)(\sigma\sigma') = l\sigma'$ of $\phi_\uparrow(l, \varphi)$ is derivable and $|\widetilde{\vdash}(s_M, \phi_\uparrow(l, \varphi))| = n + 1$. This contradicts the meaning of $i_{max\_\rho}(\varphi, n)$. Hence, the existence of an additional instance of a literal $l$ with respect to $s_M$ is impossible and $l$ is not open-ended. □

Proposition 3.1 can be used by the reasoner to determine that a given literal is not open-ended. In this way, the open-endedness can be reduced.

Moreover, the following proposition holds:

**Proposition 3.2.** *A literal* `neg` *$l$ is derivable with respect to $s_M$ if $l$ is neither derivable nor open-ended with respect to a state model $s_M$.*

*Proof.* If $l$ is not open-ended, then all instances of $l$ are derivable. If $l$ is not derivable (i.e., not a member of the set of derivable instances of $l$), then all instances of $l$ cannot hold with respect to the domain model at hand. Thus, `neg` $l$ holds. □

Proposition 3.2 constitutes a rule that is added to the state model and represented by the following definite clause:

$$d(\texttt{neg}\ l) \leftarrow \nvdash d(l), \nvdash d(l^{\sqsubseteq}) \tag{GR27}$$

For example, we can now derive the statement `neg in_room(bobs_mug,office)` if `in_room(bobs_mug,kitchen)` and $i_{max\_\rho}$(`in_room(`$ground^*$`,`$var^*$`),1`) are derivable. Thus, we can now omit the domain specific rule DR2 in order to remove the redundancy without loosing derivable information.

## 3.9.2. Viewing Statements as Concepts

Statements are not just syntactical constructs. They constitute *concepts*. In this sense, basic terms (see Section 3.3) can be seen as *atomic concepts*. The semantics of the concept that is constituted by a basic term is defined by an *interpretation function* $\mathcal{I}$ that maps an n-nary basic term to a set of n-tuples. More precisely, the interpretation function is defined as follows:

**Definition 3.13** (interpretation function)**.** The *interpretation function* $\mathcal{I}$ maps an n-nary basic term $f(X_1, \ldots, X_n)$ with respect to a state model $s_M$ to a set of n-tuples $f(X_1, \ldots, X_n)^{\mathcal{I}}$ such that the following holds:

$$\{(X_1\sigma, \ldots, X_n\sigma) \mid f(X_1, \ldots, X_n) \text{ is derivable w.r.t. } s_M \text{ and } \sigma\} \subseteq f(X_1, \ldots, X_n)^{\mathcal{I}}$$

If it can be derived from a state model $s_M$ that $(a_1, \ldots, a_n)$ is a member of $f(X_1, \ldots, X_n)^{\mathcal{I}}$, then $f(X_1, \ldots, X_n)$ is defined to be derivable with respect to $s_M$ and the substitution $\{X_1/a_1, \ldots, X_n/a_n\}$.

For a basic term $f(X_1, \ldots, X_n)$, the set of tuples $f(X_1, \ldots, X_n)^{\mathcal{I}}$ contains a tuple $(X1, \ldots, X_m)\sigma$ for each instance $f(X_1, \ldots, X_n)\sigma$ that holds in the current situation. Due to the fact that the state model is open-ended, only the subset

$$\{(X_1\sigma, \ldots, X_n\sigma) \mid f(X_1, \ldots, X_n) \text{ is derivable w.r.t. } s_M \text{ and } \sigma\}$$

of $f(X_1, \ldots, X_n)^{\mathcal{I}}$ can be derived from a state model $s_M$.

For example, consider a situation where an agent is only aware of the existence of two mugs. These mugs are called Bob's mug and Peter's mug. Thus, the set of derivable instances of the basic term `mug(X)` with respect to the state model of the agent $s_M$ is:

$$\widetilde{\vdash}(s_M, \texttt{mug(X)}) = \{\texttt{mug(bobs\_mug)}, \texttt{mug(peters\_mug)}\}.$$

For this example, the following holds:

$$\{(\texttt{bobs\_mug}), (\texttt{peters\_mug})\} \subseteq \texttt{mug(X)}^{\mathcal{I}}$$

Atomic concepts can be extended to concepts constituted by literals. Let $\mathcal{L}_{const}$ be the set of all constant symbols. The semantics of the concept constituted by a

negative literal `neg` $p(X_1, \ldots, X_n)$ is defined as follows:

$$(\texttt{neg}\ p(X_1, \ldots, X_n))^{\mathcal{I}} = (\mathcal{L}_{const})^n \backslash (p(X_1, \ldots, X_n))^{\mathcal{I}} \tag{3.2}$$

Viewing literals as concepts enables it to support the definition of relations on the level of concepts. The proposed state model supports the definition of the fact that a concept is the *subconcept* of another concept and that two concepts are *disjoint*. The subconcept relation is defined as follows:

**Definition 3.14** (subconcept)**.** The concept constituted by a literal $l$ is said to be a *subconcept* of the concept constituted by a literal $l'$ (denoted as $l \sqsubseteq l'$) iff $l^{\mathcal{I}}$ is a subset of $l'^{\mathcal{I}}$.

The subconcept relation can be explicitly defined as part of the state model specification. It can only be defined for literals that have the same arity. Let $X_i\ (1 \leqslant i \leqslant n)$ be variables and $p(X_1, \ldots, X_n)$ and $p'(X_1, \ldots, X_n)$ be literals. The fact that a concept that is constituted by a literal $p(X_1, \ldots, X_n)$ is a subconcept of the concept constituted by a literal $p'(X_1, \ldots, X_n)$ is specified by constructs of the following form:

$$p(X_1, \ldots, X_n) \sqsubseteq_{def} p'(X_1, \ldots, X_n) \tag{3.3}$$

For example, the fact that a cup is a container can be defined by the following construct:

$$\texttt{cup(X)} \sqsubseteq_{def} \texttt{container(X)}$$

The explicit definition of subconcept relations can be used by the reasoner via using the following rule:

$$d(l \sqsubseteq l') \leftarrow d(l \sqsubseteq_{def} l') \tag{GR28}$$

Furthermore, the state model can exploit knowledge about subconcept relations based on the following proposition:

**Proposition 3.3.** *Let $p(X_1, \ldots, X_n)$ be subconcept of $p'(X_1, \ldots, X_n)$ and $s_M$ be a state model. If $p(X_1, \ldots, X_n)$ is derivable with respect to $s_M$ and $\sigma$, then $p'(X_1, \ldots, X_n)$ is also derivable with respect to $s_M$ and $\sigma$.*

*Proof.* Let $p(X_1, \ldots, X_n)$ be derivable with respect to $s_M$ and $\sigma$. According to Definition 3.13, this implies that $(X_1\sigma, \ldots, X_n\sigma)$ is a member of $p(X_1, \ldots, X_n)^{\mathcal{I}}$. Based on Definition 3.14, we can infer that $(X_1\sigma, \ldots, X_n\sigma)$ is also a member of $p'(X_1, \ldots, X_n)^{\mathcal{I}}$. Thus, based on Definition 3.13, we can deduce that $p'(X_1, \ldots, X_n)$ is derivable with respect to $s_M$ and $\sigma$.

□

Proposition 3.3 constitutes the theoretical background that enables the definition of the following rule:

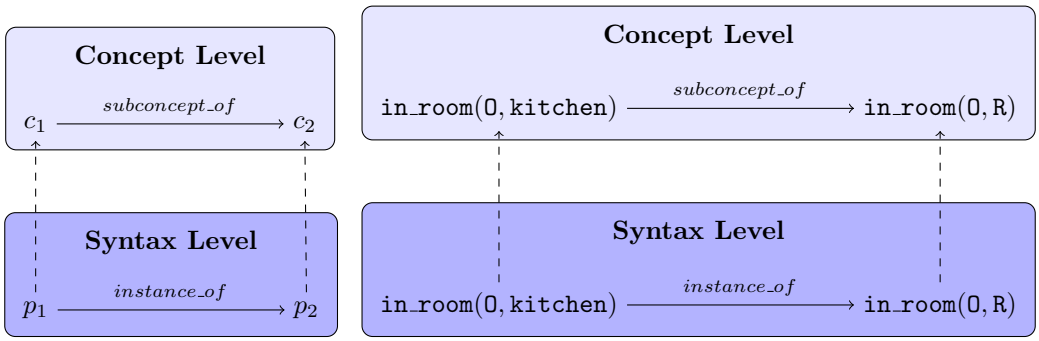



Figure 3.3.: If a literal $p_1$ is an instance of a literal $p_2$ on the syntactic level, then the concepts that is constituted by $p_1$ is a subconcept of the concept constituted by $p_2$ (a). For example, `in_room(0,kitchen)` is an instance of `in_room(0,Room)` on the syntactical level and a subconcept of `in_room(0,Room)` on the conceptual level (b).

$$d(l) \leftarrow d(l' \sqsubseteq l), d(l') \tag{GR29}$$

In other words, a literal is derivable if there is a subconcept that is derivable. Moreover, the subconcept relation can be viewed as related to the instance-of relation. As illustrated in Figure 3.3, the instance-of relation on the syntactical level implies a subconcept-of relation on the conceptual level.

Furthermore, based on the semantics of concepts that are constituted by negative literals, the following can be proposed:

**Proposition 3.4.** *Let $l, l'$ be n-nary basic term. If $l$ constitutes a subconcept of $l'$ ($l \sqsubseteq l'$), then* `neg` $l'$ *constitutes a subconcept of* `neg` $l$ *(*`neg` $l' \sqsubseteq$ `neg` $l$*).*

*Proof.* If $l$ is a subconcept of $l'$, then (according to Definition 3.14) $l^{\mathcal{I}}$ is a subset of $l'^{\mathcal{I}}$. Thus, $(\mathcal{L}_{const})^n \backslash l'^{\mathcal{I}}$ must be a subset of $(\mathcal{L}_{const})^n \backslash l^{\mathcal{I}}$. Therefore, `neg` $l'$ must constitute a subconcept of `neg` $l$. □

Proposition 3.4 can be exploited by the reasoner in the form of the following rule:

$$d(\texttt{neg}\ l' \sqsubseteq \texttt{neg}\ l) \leftarrow d(l \sqsubseteq l') \tag{GR30}$$

Some literals cannot be instantiated with the same substitution. For instance, if we assume that a mug cannot be a table, then `mug(a)` and `table(a)` cannot both hold for the same `a`. The concepts constituted by these literals are said to be *disjoint*. In general, concepts are called disjoint as defined by following definition:

**Definition 3.15** (disjoint)**.** The concepts that are constituted by two literals $l, l'$ are called *disjoint* iff $l^{\mathcal{I}}$ and $l'^{\mathcal{I}}$ are disjoint (i.e., $l^{\mathcal{I}} \cap l'^{\mathcal{I}} = \varnothing$).

Like for the definition of subsumption relations, it is possible to explicitly define that two literals are *disjoint*. The fact that the concepts constituted by the literals $p(X_1, \ldots, X_n)$ and $p'(X_1, \ldots, X_n)$ are disjoint can be defined by constructs of the following form:

$$p(X_1, \ldots, X_n) \sqcap_{def} p'(X_1, \ldots, X_n) \tag{3.4}$$

The following rule is added to the state model in order to support the definition of disjoint concepts:

$$d(l \sqcap l') \leftarrow d(l \sqcap_{def} l') \tag{GR31}$$

Moreover, based on the knowledge about disjoint concepts, we can make the following proposition:

**Proposition 3.5.** *Let $t_i (1 \leqslant i \leqslant n)$ be a term. The negation* neg $p(t_1, \ldots, t_n)$ *of a literal $p(t_1, \ldots, t_n)$ is derivable with respect to a state model $s_M$ and a substitution $\sigma$ if there is a disjoint literal $p'(t_1, \ldots, t_n)$ (i.e., $p(t_1, \ldots, t_n) \sqcap p'(t_1, \ldots, t_n)$) that is derivable with respect to $s_M$ and $\sigma$.*

*Proof.* If $p'(t_1, \ldots, t_n)$ is derivable with respect to $s_M$ and $\sigma$, then $(t_1\sigma, \ldots, t_n\sigma)$ is a member of $p'(t_1, \ldots, t_n)^{\mathcal{I}}$. It follows from Definition 3.15 that $(t_1\sigma, \ldots, t_n\sigma)$ is not a member of $p(t_1, \ldots, t_n)^{\mathcal{I}}$. Therefore, $(t_1\sigma, \ldots, t_n\sigma)$ must be a member of $(\mathcal{L}_{const})^n \backslash p(t_1, \ldots, t_n)^{\mathcal{I}}$. It follows from 3.2 that $(t_1\sigma, \ldots, t_n\sigma)$ is also a member of $(\texttt{neg}\ p(t_1, \ldots, t_n))^{\mathcal{I}}$. Hence, neg $p(t_1, \ldots, t_n)$ is derivable with respect to $s_M$ and $\sigma$. □

Proposition 3.5 can be exploited by the reasoner in form of the following rule:

$$d(\texttt{neg}\ l) \leftarrow d(l \sqcap l'), d(l') \tag{GR32}$$

## 3.10. Implementation Issues

The proposed state model is implemented in the object-oriented logic programming language *Logtalk* (Moura, 2003) using SWI-Prolog as the back-end compiler. Therefore, the definite clause programs constituted by a state model are interpreted according to the execution model of Prolog. The execution model of Prolog can be described in the following way by means of viewing the execution process as an abstract interpreter:

> Prolog's execution mechanism is obtained from the abstract interpreter by choosing the leftmost goal instead of an arbitrary one and replacing the non-deterministic choice of a clause by sequential search for a unifiable clause and backtracking (Sterling and Shapiro, 1994, page 120).

According to Sterling and Shapiro (1994), Prolog uses a stack scheduling policy. Goals are maintained as a stack. Prolog systems pop the top goal for reduction and push the derived goals onto the stack. Nondeterminism is simulated by means of choosing the first definite clause whose head unifies with a goal. If no unifiable clause can be found, then the reasoner backtracks to the last choice point.

Based on the execution mechanism of Prolog, statements are interpreted like definite goals by Prolog from left to right, using sequential search and backtracking. In other words, the execution mechanism is similar to Prolog's execution mechanism. The key difference is that the proposed state model is open-ended, whereas Prolog is based on the closed-world assumption.

However, the fact that the execution model of the proposed state model is based on Prolog's execution model has several advantages:

1. The execution model of Prolog is known to be an excellent trade-off between the abstract model from mathematical logic and an efficient implementation.

2. Prolog's execution model is well-understood and has been successfully used in various applications.

3. A lot of literature that introduces and defines the execution model is available. This eases the learning and understanding of the proposed state model.

4. The performance of the proposed state model and the corresponding reasoning processes heavily benefit from high performance Prolog systems. Basic operations like unification and backtracking do not have to be reimplemented.

A more detailed description of Prolog's execution model can be found in (Sterling and Shapiro, 1994, Section 6.1).

## 3.11. Experimental Evaluation

The performance of the open-ended reasoning process is evaluated in this section. An implementation of the reasoner that is based on the *closed world assumption* (CWA) (i.e., an implementation that only generates derivable but not possibly-derivable instances of a statement) serves as the baseline. Two different studies were conducted for the purpose of getting deeper insights into the performance characteristics of the proposed state model and corresponding reasoning processes. The first study (see Section 3.11.1) analyzes the performance for an increasingly large conjunction of independent literals, whereas the second study (see Section 3.11.2) evaluates the behavior of the reasoner for a set of preconditions from four different domain models.

These two studies evaluate the state model and the reasoner in isolation. The experimental evaluation of the whole plan-based control system is described in Chapter 9.

### 3.11.1. Independent Solutions

If two or more literals of a conjunction contain one or more identical variables, then this imposes additional constraints. For example, the statement

$$\texttt{bottle(X)} \wedge \texttt{red(X)} \wedge \texttt{in\_room(X,kitchen)} \tag{3.5}$$

describes a red bottle that is in the kitchen. The fact that all three literals contain the variable `X` restricts the number of possible solutions. For example, let us assume that a state model contains information about three bottles, but only one bottle is known to be red and in the kitchen. In such a situation, there is only one derivable solution of the aforementioned statement. In contrast, if the literals of a conjunction are independent (i.e., have no common variables), then the number of possible solutions usually increases exponentially with respect to the number of literals. For example, consider the following statement:

$$\texttt{bottle}(\texttt{X}_\texttt{1}) \wedge \ldots \wedge \texttt{bottle}(\texttt{X}_\texttt{n}) \tag{3.6}$$

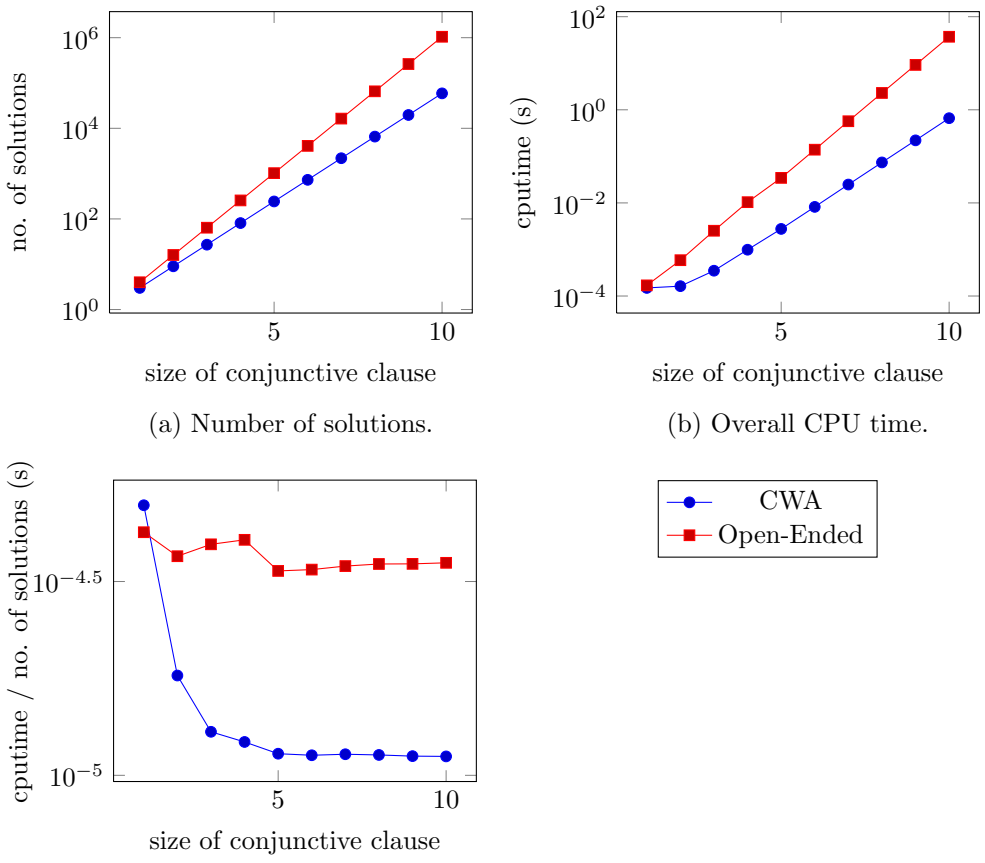
It is assumed that $\texttt{X}_\texttt{i}$ is not equal to $\texttt{X}_\texttt{j}$ iff `i` is not equal to `j`. If $m$ instances of a literal `bottle(X)` are derivable with respect to the state model at hand, then $m^n$ instances of the statement 3.6 are derivable, since every possible combination of derivable bottles constitutes a valid instance. Thus, these kinds of statements scale exponentially to the statement length for a typical CWA-based reasoner. The question addressed here is: How does the open-ended reasoner described in this chapter scale to such a statement?

One beneficial property of the proposed open-ended reasoner is that it can represent a possible infinite number of possible state model extensions by means of a single, abstract solution. For example, if the statement `bottle(X)` is possibly-derivable with respect to the set of open-ended literals $\{\texttt{bottle(X)}\}$, then this single (possibly-derivable) solution represents all solutions that contain an additional bottle. Therefore, the open-ended reasoner at most generates one additional solution for a literal (i.e., a solution with a non-empty set of open-ended literals) compared to the CWA-based reasoner. More precisely, the following proposition can be made:

**Proposition 3.6.** *If $n$ instances of a literal $l$ are derivable with respect to a state model $s_M$, then at most $n + 1$ instances of $l$ are possibly-derivable with respect to $s_M$.*

*Proof.* The correctness of this proposition follows from the fact that the reasoner only generates instances of a literal for derivable instances. If a literal is only possibly-derivable, then no substitutions are applied. Therefore, there can at most be one possibly-derivable instance of a literal $l$ with a non-empty set of open-ended literals (i.e., the set $\{l\}$). Hence, there can at most be one additional instance which is possibly-derivable, but not derivable. □

This proposition implies that if $m^n$ solutions for the statement 3.6 can be generated by a CWA-based reasoner, then at most $(m + 1)^n$ solutions can be generated




(a) Number of solutions.

(b) Overall CPU time.

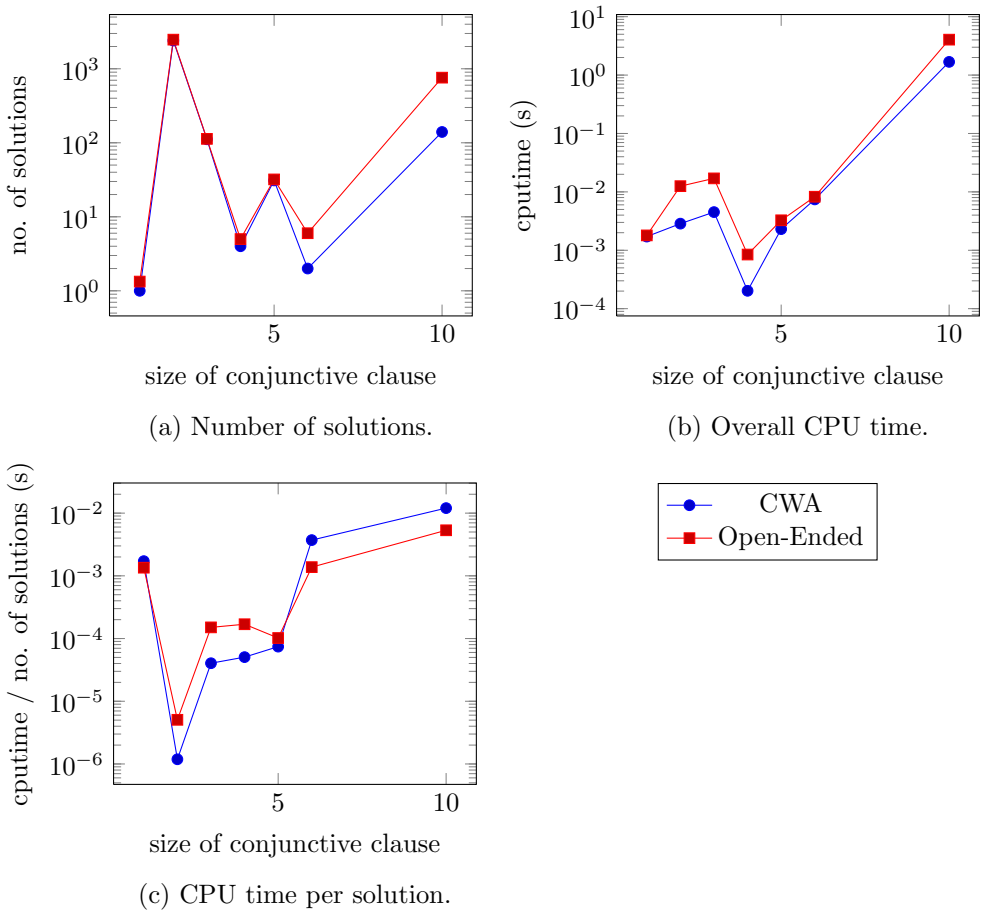
(c) CPU time per solution.

Figure 3.4.: Performance characteristics (number of solutions, CPU time, and CPU time per solution) of the proposed open-ended reasoner for an increasingly large conjunction of independent literals using a CWA-based implementation as the baseline.

by the open-ended reasoner described in this chapter.

The CWA-based and open-ended version of the reasoner are experimentally evaluated for the statement 3.6. There are three derivable and four possibly-derivable instances of a literal `bottle(X)`. Thus, for a statement with $n$ literals, the CWA-based reasoner can generate $3^n$ and the open-ended reasoner $4^n$ solutions. The experimental results are show in Figure 3.4. Figure 3.4a shows that the number of solutions increases—as expected—exponentially for both reasoners. The CPU time necessary for the generation of all plans scales similar to the number of solutions (see Figure 3.4b). The CPU time is mainly affected by the number of solutions. For larger statements, the CPU time necessary for a single solution only changes slightly for the CWA-based as well as for the open-ended reasoner (see Figure 3.4c).

In summary, we can conclude that the open-ended reasoner does not scale considerably worse than a CWA-based reasoner to an increasingly large conjunction of




(a) Number of solutions.

(b) Overall CPU time.

(c) CPU time per solution.

Figure 3.5.: Performance characteristics (number of solutions, CPU time, and CPU time per solution) of the proposed open-ended reasoner for an increasingly large conjunction of literals used as preconditions in the *robot* domain model. A CWA-based implementation of the reasoner is used as the baseline.

independent literals.

### 3.11.2. Domain Model Preconditions

For the purpose of analyzing statements that are actually used in existing domain models, the performance of the open-ended reasoner is evaluated for a set of increasingly complex preconditions from four different domain models. The first domain model is called *robot*. It is an extended version of the domain model that was used for the experiments with the physical service robot described in Section 9.1. The domain model is equal to the largest domain model used for the simulation-based experiments. It is described in more detail in Section 9.2.2. Small instances of the used domain model specifications can be found in Appendix B.

Additionally, three well-known AI planning domains are used for this study. The *depots* and *rovers* domains from the third international planning competition as well as the well-known *blocks world* domain were used. The depots and rovers domain are described in Long and Fox (2003). The blocks world domain was initially used by Winograd (1972) as a test environment for his program. Since then, it has widely been used as a testbed in the AI planning community.

The domain model instance used for the experiments contains 1758 facts for the rovers, 880 facts for depots, and 2020 facts for the blocks world domain.

#### Real-World Service Robotic Domain

The experimental results for the service robotic domain are shown in Figure 3.5. Figure 3.5 shows that the open-ended reasoner and the CWA-based baseline show a very similar behavior. The open-ended reasoner generates a bit more solutions (see Figure 3.5a) and needs a bit more CPU time (see Figure 3.5b) for the robot domain. For statements of the length one to five, the open-ended reasoner also needs a bit more CPU time for a single solution than the baseline. For statements of the length six to ten, however, the CPU time per solution is even a bit less than the baseline.

#### Blocks World Domain

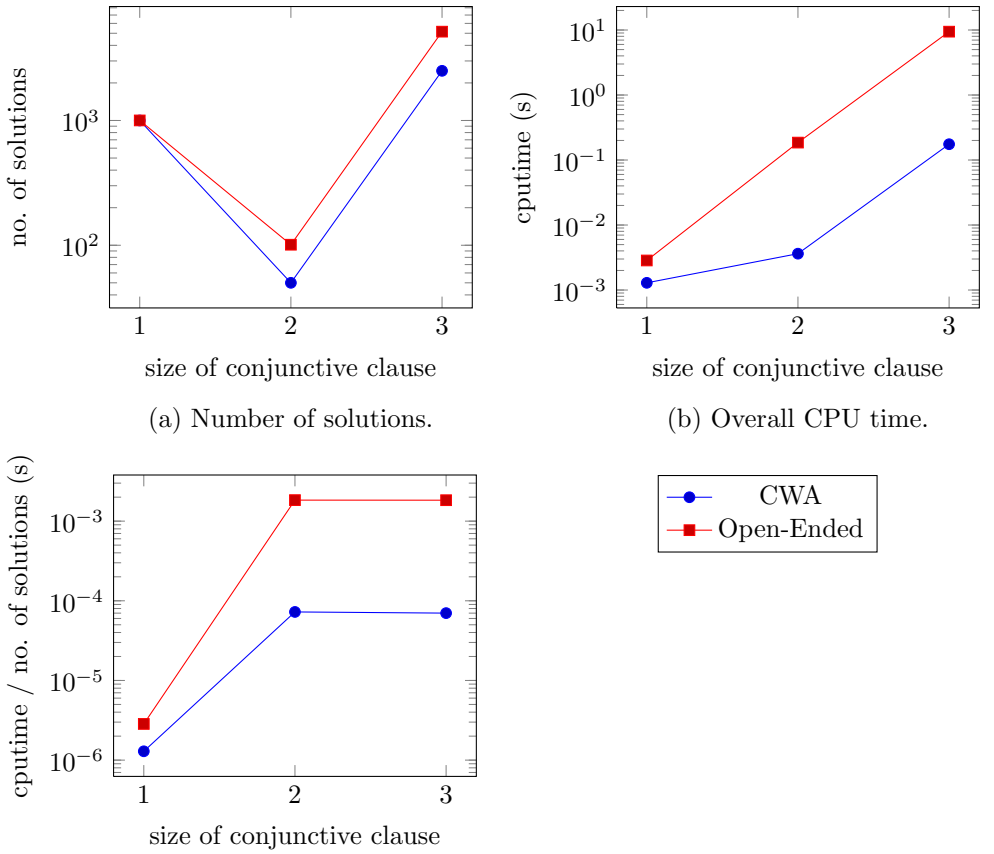
Figure 3.6 shows the experimental results for the blocks world domain. The open-ended implementation approximately generates up to two times more solutions than the baseline and needs up to two orders of magnitude more CPU time. However, the qualitative characteristics of the number of solutions (see Figure 3.6a), the overall CPU time (see Figure 3.6b), and the CPU time per solution (see Figure 3.6c) are very similar for the open-ended reasoner and the CWA-based baseline.

#### Depots Domain

The experimental results for the depots domain are shown in Figure 3.7. Compared to the baseline, the open-ended reasoner shows a similar trend for the number of solutions (see Figure 3.7a), the overall CPU time (see Figure 3.7b), and the CPU time per solution (see Figure 3.7c) for statements of the size one to three. In contrast, it scales worse for statements of length three to statements of length five.

#### Rovers Domain

Figure 3.8 shows the experimental results for the rovers domain. The number of solutions shows a similar trend like the baseline (see Figure 3.8a). The qualitative characteristics of the overall CPU time is similar to the baseline for statements of length one to three, but diverges from statement of length three to five; the CPU time for the open-ended reasoner increases, whereas the CPU time for the CWA-based baseline decreases from statements of length three to five (see Figure 3.8b). The CPU time for a single solution increases for the open-ended reasoner and decreases




(a) Number of solutions.

(b) Overall CPU time.

(c) CPU time per solution.

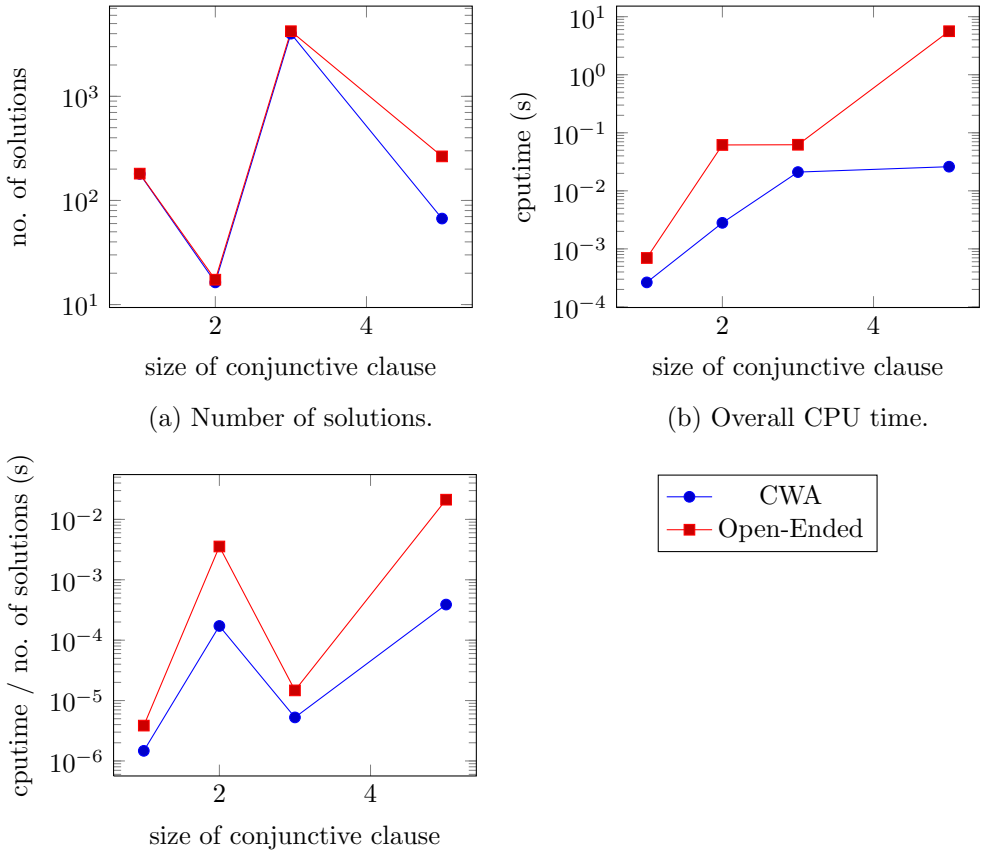
Figure 3.6.: Performance characteristics (number of solutions, CPU time, and CPU time per solution) of the proposed open-ended reasoner for an increasingly large conjunction of literals used as preconditions in the *blocks world* domain model. A CWA-based implementation of the reasoner is used as the baseline.

for the baseline for statements of length one to two. For statements of length two to five, however, both reasoners show a similar qualitative behavior (see Figure 3.8c).

### 3.11.3. Summary

The CWA-based reasoner usually constitutes a lower bound for the number of solutions, the overall CPU time, and the CPU time per solution. Table 3.1 shows the average characteristics of the open-ended reasoner relative to the CWA-based reasoner. On average, the open-ended reasoner generates 3.89 times as many solutions, needs 29.95 times as many CPU time for the whole reasoning process, and needs 9.36 times as many CPU time for a single solution as the CWA-based reasoner. In other words, the CWA-based reasoner generates less solutions and is faster.

However, the qualitative characteristics of the open-ended reasoner is often very



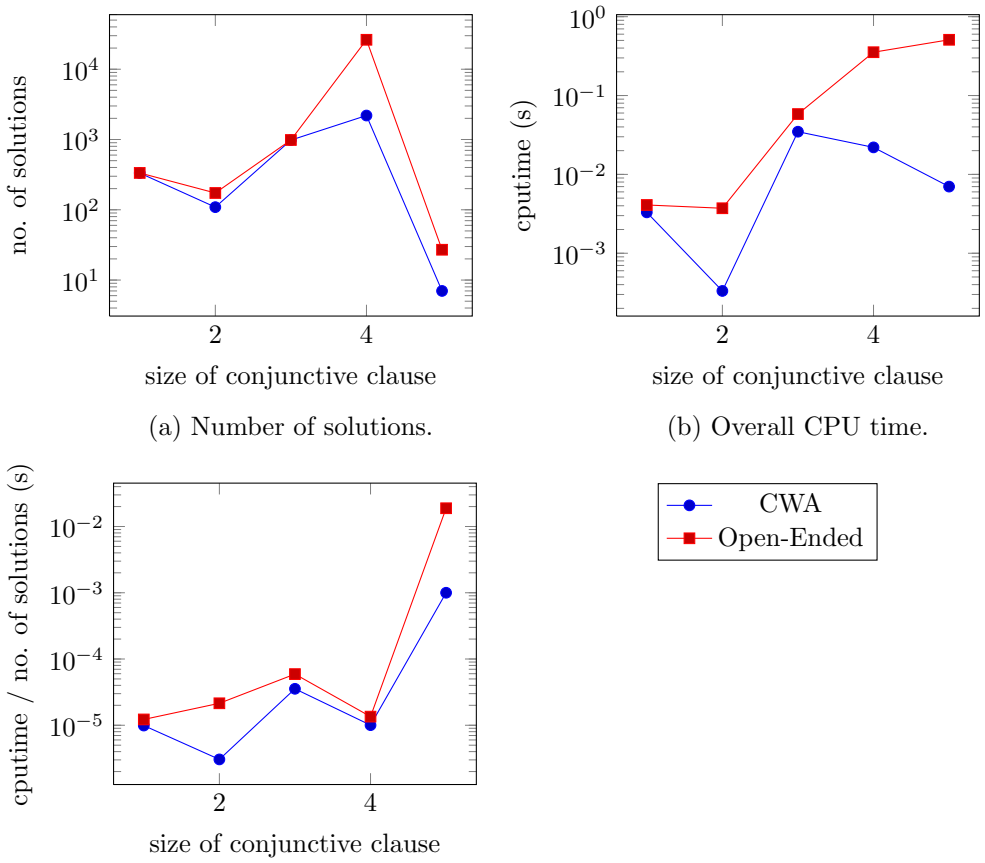

(a) Number of solutions.

(b) Overall CPU time.

(c) CPU time per solution.

Figure 3.7.: Performance characteristics (number of solutions, CPU time, and CPU time per solution) of the proposed open-ended reasoner for an increasingly large conjunction of literals used as preconditions in the *depots* domain model. A CWA-based implementation of the reasoner is used as the baseline.

similar to the CWA-based baseline. Thus, the CWA-based reasoner is faster, but often does not scale significantly better to more complex statements.




(a) Number of solutions.

(b) Overall CPU time.

(c) CPU time per solution.

Figure 3.8.: Performance characteristics (number of solutions, CPU time, and CPU time per solution) of the proposed open-ended reasoner for an increasingly large conjunction of literals used as preconditions in the *rovers* domain model. A CWA-based implementation of the reasoner is used as the baseline.

| **Domain** | **solutions** | **CPU time** | **CPU time per solution** |
|---|---|---|---|
| robot | 1.85 | 2.62 | 2.04 |
| blocks world | 7.69 | 35.75 | 9.19 |
| depots | 1.74 | 60.82 | 20.17 |
| rovers | 3.89 | 20.59 | 6.03 |
| all (average) | 3.79 | 29.95 | 9.36 |

Table 3.1.: Average number of solutions, CPU time and CPU time per solution for the open-ended reasoner divided by the values for the CWA-based implementation.

## 3.12. Discussion and Related Work

This chapter has introduced a new state model and reasoning system. The state model is open-ended in the sense that it is not assumed that a complete model of the represented domain is available. It provides the basic concepts that enable the reasoning about possible extensions of the state model at hand. In particular, the notion of a possibly-derivable statement enables it to determine extensions that are relevant with respect to a statement such that each extension implies that an additional instance of the statement can be derived.

The proposed state model constitutes a definite program (i.e., a logic program). The definite program constituted by the generic rules can be seen as the program of a reasoner. Due to the fact that the (underlying) definite-clause-reasoning is *undecidable* (Brachman and Levesque, 2004), the reasoning processes of the state model are in general also undecidable. The reasoner provides, together with the activities model described in Chapter 4, the basic reasoning capabilities that are exploited by the new hierarchical planning system introduced in Chapter 5.

Various knowledge representations schemes for AI planning have been developed in the last decades. The *classical representation scheme* (Ghallab et al., 2004) is linked to the representation scheme used by the very influential STRIPS (Fikes and Nilsson, 1971) planning system. The Action Description Language (ADL) (Pednault, 1988, 1989) is a trade-off between the expressiveness of general logic-based representations and the computational complexity of corresponding reasoning processes. The PDDL (Ghallab et al., 1998) planning language and extensions thereof are used for international planning competitions and are supported by a multitude of planning systems. The idea of using *state variables* instead of logical atoms was introduced in the SAS planner (Bäckström and Klein, 1991; Bäckström, 1992; Bäckström and Nebel, 1995; Jonsson and Bäckström, 1998). All the aforementioned representation schemes are based on the closed-world assumption (Ghallab et al., 2004). Thus, they are inappropriate for the representation of incomplete, open-ended models.

The state model presented in this chapter has similarities with the possible-worlds model proposed by Fagin et al. (1995). Fagin et al. (1995) provide a general framework for reasoning about knowledge and possible worlds. However, in contrast to the work of Fagin et al. (1995), the state model proposed in this work is more tailored to the integration into a plan-based control system. In particular, the introduced notion of a possibly-derivable statement is a key, unique concept that enables an agent to efficiently reason about extensions of a state model with respect to certain statement.

Most existing automatic theorem proving or knowledge representation and reasoning systems, including the domain models of AI planning systems, do not systematically analyze failed inferences or queries. The only known exception is the "WhyNot" tool of PowerLoom (Chalupsky and Russ, 2002) which tries to generate a set of plausible partial proofs for failed queries. Nevertheless, "WhyNot" is rather a debugging tool that tries to generate human readable explanations that describe

why the overall reasoning process failed. Therefore, this approach is not adequate for the objectives of this work.

Exploiting local closed-world assumptions is also featured by PowerLoom (Chalupsky et al., 2010) and has also been proposed by Etzioni et al. (1997).

The approach of Dornhege et al. (2009) also makes it possible to integrate external components into the planning process. However, integration is not done autonomously (i.e., by reasoning on the need to acquire information from external sources), but predefined in the domain description.

Chapter 4

# Activities Model

*Although the representation we select will have inevitable consequences for how we see and reason about the world, we can at least select it consciously and carefully, trying to find a pair of glasses appropriate for the task at hand.* (Davis et al., 1993)

## Contents



In addition to knowledge about its environment, an agent needs information about its own *activities* for the purpose of planning its future course of action. Information about the activities of the agent is represented by the *activities model*. Together with

the state model, the activities model constitutes the domain model that is used by the proposed plan-based control system. The purpose of this chapter is to describe and define this activities model. It can be seen as an adaption and extension of existing activities models of HTN planning systems. The proposed activities model is particularly inspired by the model described in (Ghallab et al., 2004, Chapter 11) and the underlying activities models of SHOP (Nau et al., 1999) and SHOP2 (Nau et al., 2003).

Compared to existing activities models, the proposed model provides several new features that enable the reasoning about active knowledge acquisition. It provides the basic knowledge representation and reasoning techniques that permit the automatic integration of relevant knowledge acquisition tasks into the overall planning and execution process.

## 4.1. Preliminaries

Some aspects of the agent's activities need to be evaluated during the planning and execution process. For example, the cost of approaching a goal pose depends—besides other aspects—on the current position of the agent. Thus, one needs to define the cost as a function which is evaluated during the planning and execution process. The ACogDM domain model features *ACog expressions* for the purpose of defining functions that can be evaluated at runtime. These expressions are defined as follows:

**Definition 4.1** (ACog Expression)**.** An *ACog expression* is a term of the form $eval(t, st)$ where $t$ is a term and $st$ is a statement.

The *value* $eval(t, st)^{v(s_M)}$ of an expression $eval(t, st)$ with respect to a state model $s_M$ is defined as follows:

$$eval(t, st)^{v(s_M)} := \begin{cases} t\sigma & \text{if } s_M \vdash_\sigma \textbf{once } st \\ \text{failure} & \text{otherwise} \end{cases} \tag{4.1}$$

Thus, ACog expressions are evaluated using the reasoning services of the state model. Due to the fact that the value of an expression is evaluated using the **once** meta-predicate, the reasoner only returns one value. Therefore, one can actually view an expression as a function that maps a state model and an expression to a corresponding value.

## 4.2. Basic Components

The basic components and underlying concepts of the activities model are introduced and explained in this section.

### 4.2.1. Task

In contrast to classical AI planning, HTN planning proceeds via recursively decomposing a *task* into a number of subtasks until the level of *primitive tasks* is reached. Therefore, each HTN planning system needs a definition of a task. For this work, the definition of a *task* is borrowed from (Ghallab et al., 2004, page 231). A task is defined as an expression of the form $t(r_1, \ldots, r_k)$ such that $t$ is a task symbol and $r_i (1 \leqslant i \leqslant k)$ are terms. An task is called *primitive* iff it can directly (i.e., without task planning) be executed by the corresponding agent. Otherwise, a task is called *nonprimitive*.

### 4.2.2. Planning Operator

Like in classical planning, *planning operators* represent *primitive actions* that can be directly (i.e., without planning) executed by an agent. Primitive actions are viewed as atomic. In practice, primitive actions are usually not atomic and can make use of dedicated planning systems. However, from the AI planning perspective, this internal behavior is abstracted away. In the context of plan-based agent control, the role of primitive actions is twofold:

1. They are the basic building blocks of plans. A planner uses—besides other information—the knowledge encapsulated in planning operators for the purpose of generating plans.

2. For each primitive action, there is a corresponding agent control program that can be executed by the agent. Primitive actions represent these control programs.

The proposed planning system considers the effect of an action in terms of a set of literals that are removed and a set of literals that are added to the state model after the execution. Additionally, every planning operator has a corresponding *cost function*, that defines how expensive it is to execute an instance of the action. Formally, a planning operator is defined as follows:

**Definition 4.2** (planning operator)**.** A *planning operator* is a 5-tuple

$$o := (o_{name}, o_{cond}, o_{del}, o_{add}, o_{cost})$$

in which the elements are described as follows:

- $o_{name}$ is a term that denotes the name of the operator.
- $o_{cond}$ is a statement that denotes the precondition of the operator.
- $o_{del}$ is a set of literals $L$ such that all members of $\{d(l) | l \in L\}$ need to be removed from the state model after the execution of the operator. All facts of a state model that are unifiable with one of the members of $o_{del}$ are removed.

- $o_{add}$ is a set of literals $L$ such that all members of $\{d(l)|l \in L\}$ need to be added to the state model after the execution of the operator.
- $o_{cost}$ is an ACog expression that represents the expected cost of the operator.

Like in classical planning, any ground instance of a planning operator is called an *action* (Ghallab et al., 2004, Definition 2.6):

**Definition 4.3** (action)**.** A ground planning operator is called an *action*.

Moreover, we say that an action

$$a := (a_{name}, a_{cond}, a_{del}, a_{add}, a_{cost})$$

*accomplishes* a ground primitive task $t$ iff $a_{name} = t$.

### 4.2.3. HTN Method

In contrast to classical planning, hierarchical planning proceeds via successively decomposing *nonprimitive tasks* into subtasks until the level of primitive tasks is reached.

Like other hierarchical planning systems, the activities model contains additional domain specific information in form of *Hierarchical Task Network (HTN) methods* (Ghallab et al., 2004). HTN methods prescribe how a task can be decomposed into a number of subtasks. A method is defined as follows:

**Definition 4.4.** A *method* is a 6-tuple

$$m := (m_{task}, m_{cond}, m_{del}, m_{add}, m_{tasks}, m_{cost})$$

where the elements are defined as follows:

- $m_{task}$ is a task so that the method is *relevant* for a task $t$ iff $t$ is an instance of $m_{task}$.
- $m_{cond}$ is a statement that represents the precondition of the method.
- $m_{del}$ is a set of literals $L$ such that all members of $\{d(l)|l \in L\}$ need to be removed from the state model after the task is performed according to the definition of the method. All facts of a state model that are unifiable with one of the members of $m_{del}$ are removed.
- $m_{add}$ is a set of literals $L$ such that all members of $\{d(l)|l \in L\}$ need to be added to the state model after the task is performed according to the definition of the method.
- $m_{tasks}$ is a (totally-ordered) sequence of (sub-)tasks. If the precondition $m_{cond}$ holds, then the task $m_{task}$ can be decomposed into $m_{tasks}$.

```
method(
  % task
  move_to(R),
  % precondition
  (in_room(agent,R1) ∧ connect(R1,Door,R) ∧ open(Door) ∧
    at_pose(robot,StartPose) ∧ approach_pose(Door,GoalPose) ∧
    reachable(GoalPose,Cost)),
  % delete-set
  [in_room(R1)],
  % add-set
  [in_room(R)],
  % subtasks
  [move_to_pose(GoalPose),move_forward(500)],
  % expected cost
  eval(Cost1,Cost1 is Cost + 500)).
```

Figure 4.1.: Example specification for the task `move_to(R)`.

- $m_{cost}$ represents the expected cost of performing the task according to definition of the method.

The effect of a method is called a *high-level effect*, since the effects of a method are usually on a higher abstraction layer than the effects of planning operators. The effect of a method becomes *active* after the sequence of subtasks $m_{tasks}$ has been performed.

Similar to the simplified specification described in Section 2.1.2, methods are technically specified as atomic formulas in Prolog (Deransart et al., 1996) syntax of the following form:

$$\texttt{method}(\texttt{Task}, \texttt{Precondition}, \texttt{DeleteSet}, \texttt{AddSet}, \texttt{Subtasks}, \texttt{Cost})$$

The *i-th* argument of this formula represents the *i-th* argument of a method $m$ defined as described by Definition 4.4. For example, `Precondition` represents the precondition $m_{cond}$ for a method $m$. If a method has no high-level effects (i.e., the delete-set and add-set are empty), then the terms `Delete-Set` and `Add-Set` can be omitted.

A method for the task of moving to a room is shown in Figure 4.1. The task of moving into a room is decomposed into the task of approaching a pose which is directly in front of a corresponding door, and moving 500 millimeters forward. Approaching a pose with the mobile platform and moving 500 millimeters ahead in general does not imply that the robot is moving to a different room. It only implies moving into a different room if the pose is directly in front of a door. Generally speaking, it depends on the context. This context can be provided by an HTN method. For example, it can be provided by the method specified in Figure 4.1. The fact that a method specification can also include an effect in form of a delete

and an add-set makes it possible to derive (high-level) effects that cannot be derived from the effects of the primitive actions. For example, the fact that an agent moves to a different room by means of moving in front of a door and moving 500 millimeters ahead can be derived from the high-level effects of the method specified in Figure 4.1, but cannot be derived from the effects of its subtasks.

### 4.2.4. Plan

Any plan-based system needs a representation of plans. In the context of this work, a plan is defined as follows:

**Definition 4.5** (plan)**.** A plan is 2-tuple

$$p := (p_{tasks}, p_{actions})$$

in which the elements are described as follows:

- $p_{tasks}$ is a sequence of tasks.
- $p_{actions}$ is a sequence of actions.

A plan is called *final* iff $p_{tasks}$ is empty and called *intermediate* iff $p_{tasks}$ is non-empty.

In contrast to many other planning approaches, the notion of a plan introduced by Definition 4.5 allows it to leave some parts of the plan on an abstract level. For a plan $p$, the sequence of tasks $p_{tasks}$ can contain tasks on various abstraction levels. In contrast to the sequence of actions $p_{actions}$, the tasks in $p_{tasks}$ usually cannot be executed directly.

For example, Figure 4.2 illustrates the plans and plan transformations for the decompositions illustrated by Figure 2.3.

## 4.3. Knowledge Acquisition Task

Agents (e.g., robots) can often acquire information from a multitude of sources. These sources are called *external knowledge sources*. While submitting questions to external databases or reasoning components might be "simply" achieved by calling external procedures, submitting questions to other sources (e.g., sensors), however, involves additional planning and execution. In other words, a planner needs to be able to generate plans that acquire possibly relevant information. For the purpose of enabling the planner to generate such knowledge acquisition plans, a particular kind of task is introduced, namely a *knowledge acquisition task*.

The general idea is that knowledge acquisition tasks can be automatically generated by the plan-based control architecture if the acquisition of additional information is necessary. For instance, consider the situation illustrated by Figure 2.4 where

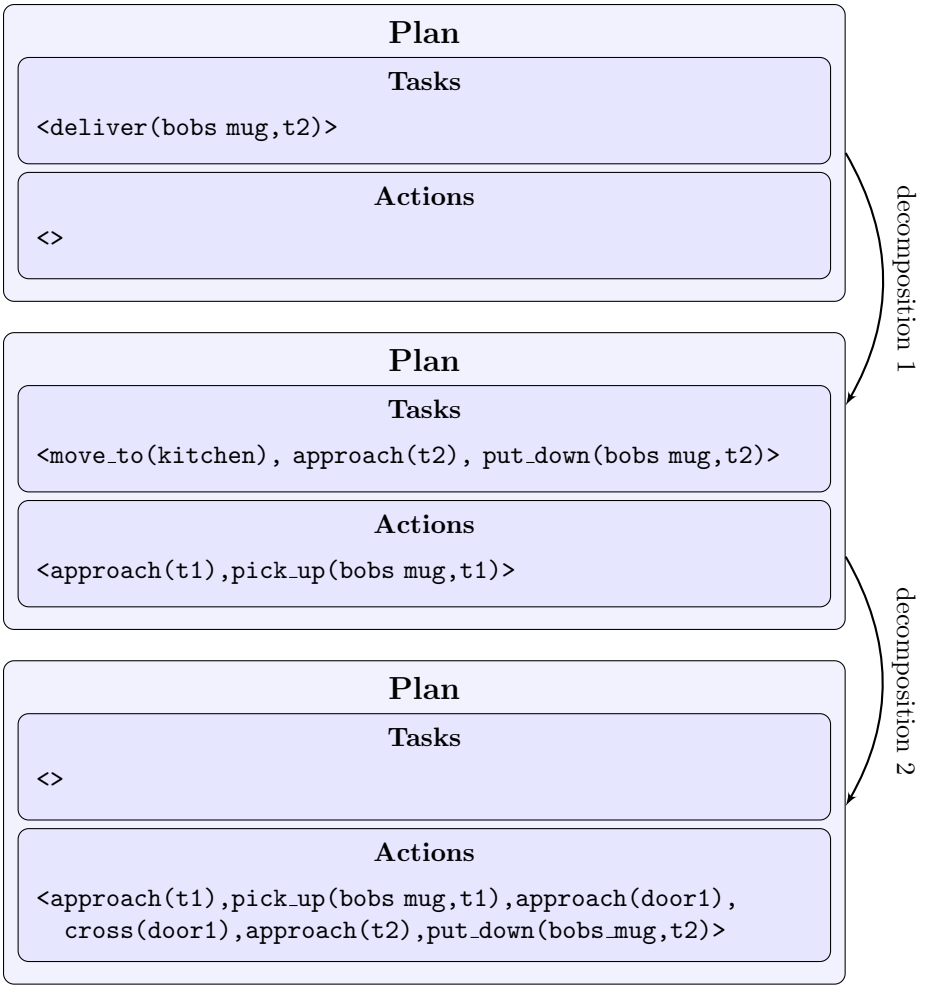



Figure 4.2.: Illustration of the plan transformations for the example illustrated by Figure 2.3.

a robot wants to perform the task `move_to(kitchen)`. As illustrated by Figure 2.4, in open-ended domains there are in principle three ways to decompose this task into a sequence of subtasks. The first decomposition results in an executable plan. The planner has sufficient information for the purpose generating this plan. Therefore, no additional knowledge acquisition tasks need to performed. In contrast, the second and third way of moving into the kitchen require the acquisition of additional information. For example, let us consider the third method instance which plans to move into the kitchen via a possibly existing additional door `D`. In this case, the robot must first find an additional (i.e., currently unknown) door that connects the kitchen with the lab and is open. However, in order to find such a door (e.g., via perception) a plan for a corresponding knowledge acquisition task has to be generated. The idea is that the plan-based control architecture automatically generates such a task and a corresponding plan if necessary. For the example at hand, the precondition of the relevant method (see Figure 2.2)

$$\texttt{in\_room(agent,R1)} \wedge \texttt{connect(R1,Door,R)} \wedge \texttt{open(Door)}$$

is possibly-derivable with respect to a substitution that replaces `R` with `kitchen` and `R1` with `lab`, and the set of open-ended literals

$$\{\texttt{connect}(\texttt{lab}, \texttt{D}, \texttt{kitchen}), \texttt{open}(\texttt{D})\}.$$

The robot must find a new instance for both open-ended literals. However, in this situation, it must be considered that one cannot independently find an open door and a door that connects the kitchen with the lab. The fact that both literals refer to the same door `D` imposes the additional constraint that a single door needs to found that connects the kitchen with the lab and is open. In this case, the literals are called *dependent*. Dependency between literals is a transitive relation. More precisely, it is defined as follows:

**Definition 4.6** (dependent literals)**.** Let $l_1, l_2$ be literals that are a member of the same set of open-ended literals and $var(l)$ denote the set of variables of a literal $l$. Two literals $l_1$ and $l_2$ are called *dependent* (denoted as $l_1 \leftrightarrow l_2$) iff one of the following holds:

- $l_1$ and $l_2$ are identical,
- $l_1$ and $l_2$ contain an identical variable ($var(l_1) \cap var(l_2) \neq \varnothing$),
- or it exists a literal $l_3$ such that $l_1$ and $l_3$, as well as $l_3$ and $l_2$ are dependent ($\exists_{l_3}\, l_1 \leftrightarrow l_3 \wedge l_3 \leftrightarrow l_2$).

The dependencies between open-ended literals need to be encoded in the specification of a knowledge acquisition task so that they can be considered by the planner.

Another thing that needs to be considered for the aforementioned example is the fact that a few instances of the open-literals are already derivable. For the literal `open(D)`, the instance `open(door1)`; and for the literal `connect(lab,D,kitchen)`, the instance `connect(lab,door1,kitchen)` as well as `connect(lab,door2,kitchen)` are already derivable. This information needs to be encoded into the definition of a corresponding knowledge acquisition task so that the agent does not try to acquire information that is already known. Taking into account these observations, a knowledge acquisition tasks is defined as follows:

**Definition 4.7.** A *knowledge acquisition task* of a state model $s_M$ has the form

$$\texttt{det}(ks, l, I, C, l_r)$$

where the components are defined as follows:

- $ks$ is the symbolic representation of an external knowledge source.
- $l$ is a literal for which a new instance should be acquired.

- $I$ is the set of all instances of $l$ that are derivable with respect to $s_M$ ($I = \widetilde{\vdash}(s_M, l)$).
- $C$ is a set of literals that are dependent on $l$. Let $L_x$ be a set of open-ended literals so that $l$ is a member of $L_x$. $C$ is defined as follows:
  $C := \{l' \mid l' \in (L_x \backslash \{l\})$ and $l'$ is dependent on $l\}$
- $l_r$ is the result of the knowledge acquisition.

The result of a knowledge acquisition task is:

- an additional instance $l\sigma$ of $l$ if such an instance can be determined,
- *impossible* if it can be determined that the existence of an additional instance of $l$ is impossible,
- or *indeterminable* otherwise. In this case the knowledge acquisition process could not provide additional information with respect to the literal $l$.

In a nutshell,

$$\texttt{det}(ks, l, I, C, l_r)$$

is the task of acquiring an instance $l\sigma$ of $l$ from the knowledge source $ks$ such that $l\sigma$ is not a member of $I$ (i.e., $l\sigma$ is not already derivable) and for all $c \in C$ an instance of $c\sigma$ is derivable.

For the aforementioned example, the task of acquiring a new instance of

$$\texttt{connect}(\texttt{lab}, \texttt{D}, \texttt{kitchen})$$

via using the laser scanner as a knowledge source is defined as follows:

$$\texttt{det}(\texttt{laser}, \texttt{connect}(\texttt{lab}, \texttt{D}, \texttt{kitchen}),$$
$$\{\texttt{connect}(\texttt{lab}, \texttt{door1}, \texttt{kitchen}), \texttt{connect}(\texttt{lab}, \texttt{door2}, \texttt{kitchen})\}, \{\texttt{open}(\texttt{D})\}, \texttt{l}_\texttt{r})$$

Due to the fact that a single knowledge source cannot provide information about arbitrary aspects of the environment, the consideration of the context of a knowledge acquisition task is optional. For example, consider the task of finding a mug which belongs to Joe, is located in the room `r1` and is not Bob's mug via using the vision system. This task would be defined as follows:

$$\texttt{det}(\texttt{vision}, \texttt{mug}(\texttt{X}), \{\texttt{mug}(\texttt{bobs\_mug})\}, \{\texttt{in\_room}(\texttt{X}, \texttt{r1}), \texttt{belongs\_to}(\texttt{X}, \texttt{joe})\}, \texttt{l}_\texttt{r})$$

```
method(
  % task
  det(vision,rel_pos(Object,Pose,X,Y),I,C,R),
  % precondition
  (at(robot,From) ∧ can_navigate(From,Pose,NavCost))
  % subtasks
  [ navigate(Pose), move_manipulator_aside,
    sense(vision,rel_pos(Object,Pose,X,Y),I,C,R)],
  % expected cost
  eval(Cost,Cost is NavCost + 1000).
```

Figure 4.3.: Example HTN method for the knowledge acquisition task `det(vision,rel_pos(Object,Pose,X,Y),I,C,R)`.

The vision system might not be able to tell the robot whether a detected mug belongs to Joe. Nevertheless, the robot can use other sources (e.g., human-robot interaction with Joe) for the purpose of determining whether a mug belongs to Joe.

An example of an HTN method for a knowledge acquisition task is shown in Figure 4.3. The specification of the delete-set and add-set is omitted, since the method has no (high-level) effects. Figure 4.3 shows the definition of an HTN method for the acquisition task of determining the relative position of an object on a table from a certain pose. Two values (`X,Y`) are sufficient to specify the relative position of an object on a table, because the height of the table is a priori known. The method shown in Figure 4.3 has been used for the experiments described in Section 9.

Every HTN method instance has an expected cost that describes how expensive it is to perform a task as described by the method. For the method shown in Figure 4.3, the cost is defined as the sum of the cost to navigate to the goal pose and the expected cost to sense the position of the object, which is the constant value 1000.

Summing up, methods for knowledge acquisition tasks enable the planner to reason about possible knowledge acquisitions, since they describe

1. what knowledge acquisitions are possible under what conditions,
2. how expensive it is to acquire information from a specific knowledge source,
3. and how to perform a knowledge acquisition task.

It might be possible that the same information can be acquired from different external knowledge sources, and the expected cost to acquire the same information can be completely different for each source. Thus, in order to acquire additional instances for each literal of a set of open-ended literals, a planner needs to decide for each literal from which knowledge source it should try to acquire an additional instance. The result of this decision process is called a *knowledge acquisition scheme*. A knowledge acquisition scheme is a set of tuples $(l, ks)$ where $l$ is a literal and

$ks$ is an external knowledge source. It represents one possible combination of trying to acquire a non-derivable instance for each open-ended literal by an adequate knowledge source.

For example, the knowledge acquisition scheme

$$\{(\texttt{on\_table(bobs\_mug)}, \texttt{vision}), (\texttt{white\_coffee(bob)}, \texttt{hri(bob)})\}$$

represents the information that the question `on_table(bobs_mug)?` should be answered with the vision system, and the query `white_coffee(bob)?` should be submitted to Bob.

As already pointed out, a knowledge acquisition scheme is the result of a decision process of the planner. If a relevant precondition is possibly-derivable with respect to a (non-empty) set of open-ended literals, then a planner needs to generate an appropriate knowledge acquisition scheme for this set of open-ended literals. For this process, literals with the interpretation model *reasoning* (see Section 3.8) need to be handled in a special way. For example, imagine a precondition is possibly-derivable with respect to the following set of open-ended literals:

$$\{\texttt{free\_ahead(Distance)}, \texttt{Distance} > 5000\}$$

The literal `free_ahead(Distance)` represents the fact that the agent can move `Distance` millimeters straight ahead without hitting an obstacle. '`Distance` $> 5000$' is evaluated by a special arithmetic reasoner and thus has the interpretation model *reasoning*. It is open-ended, since the corresponding reasoner can only handle ground literals. Moreover, it is not possible to acquire an instance of the literal '`Distance` $> 5000$' from external knowledge sources. The general idea is to first try to acquire the distance and then call the arithmetic reasoner for the purpose of checking whether the distance is more than 5000 millimeters. The same strategy is applied to all literals that have the interpretation model *reasoning*. In other words, all literals that have the interpretation model *reasoning* are excluded from the knowledge acquisition process and not added to a knowledge acquisition scheme. In this way, their evaluation is postponed until they can be evaluated by a corresponding reasoner.

How a knowledge acquisition scheme can be generated for a possibly-applicable statement is defined by Algorithm 4.1. Algorithm 4.1 partitions the set of open-ended literals into a set of literals that have the interpretation model *reasoning* ($L_{xres}$) and a set of literal that do not have the interpretation model *reasoning* ($L_{xacq}$). The algorithm ensures that the set of variables that occur in the literals with the interpretation model *reasoning* is a subset of the set of variables that occur in the remaining literals (line 7). This is necessary in order to ensure that each literal of $L_{xres}$ will be ground after the successful acquisition of an additional instance for each literal of $L_{xacq}$. For example, Algorithm 4.1 does not generate a knowledge acquisition scheme for a statement that is only possibly-derivable with respect to the following set of open-ended literals:

**Algorithm 4.1:** generate-kas($st, s_M$)

**Result**: a tuple $(kas, \sigma)$ where $kas$ is a knowledge acquisition scheme and $\sigma$ a substitution

1 $sol \leftarrow \{(L_x, \sigma) \mid st$ is possibly-derivable with respect to $s_M$, a substitution $\sigma$, and a set of open-ended literals $L_x$ $\}$;
2 nondeterministically choose any $(L_x, \sigma) \in sol$;
3 $L_{xres} \leftarrow \{l \mid l \in L_x$ and $l$ has the interpretation model $reasoning\}$;
4 $L_{xacq} \leftarrow \{l \mid l \in L_x$ and $l$ does not have the interpretation model $reasoning\}$;
5 $vars(L_{xres}) \leftarrow \{v \mid l \in L_{xres}$ and $v$ is a variable that occurs in $l\}$;
6 $vars(L_{xacq}) \leftarrow \{v \mid l \in L_{xacq}$ and $v$ is a variable that occurs in $l\}$;
7 **if** $vars(L_{xres}) \subseteq vars(L_{xacq})$ **then**
8  $kas \leftarrow \emptyset$;
9  **foreach** *element* $l \in L_{xacq}$ **do**
10  nondeterministically choose a knowledge source `ks` such that it exists a possibly-applicable method instance of the task $\texttt{det}(\texttt{ks}, \texttt{l}, \emptyset, \emptyset, \texttt{R})$;
11  $kas \leftarrow kas \cup (l, ks)$;
12  **return** $(kas, \sigma)$;

$$\{\texttt{free\_ahead}(\texttt{Distance1}), \texttt{Distance2} > 5000\}$$

The rationale behind this behavior is that the literal `'Distance2 > 5000'` would remain open-ended even if the system determines a value for `Distance1`.

However, if all variables of $L_{xres}$ constitute a subset of all variables of $L_{xacq}$ (line 7), then, for each literal in $L_{xacq}$, the algorithm nondeterministically chooses an external knowledge source such that a possibly-applicable method instance for a corresponding knowledge acquisition task exists (lines 10 - 11).

Based on Algorithm 4.1, a knowledge acquisition scheme is defined as follows:

**Definition 4.8** (knowledge acquisition scheme)**.** A *knowledge acquisition scheme* is a set of tuples $(l, ks)$ such that $l$ is an open-ended literal and $ks$ is an external knowledge source. A knowledge acquisition scheme $kas$ is called a knowledge acquisition scheme for an instance $st\sigma$ of a statement $st$ with respect to a state model $s_M$ iff $(kas, \sigma)$ can be generated by generate-kas($st, s_M$) (see Algorithm 4.1).

## 4.4. Planning Step

The term *planning step* is used in this work as an abstraction of HTN methods and planning operators (see Figure 4.4). Applying a planning step (i.e., a planning operator or a HTN method) is the basic operation of the planning process. The idea is that planning proceeds via successively applying a planning step until a sequence of initial tasks can be decomposed into a sequence of primitive tasks or no plan can

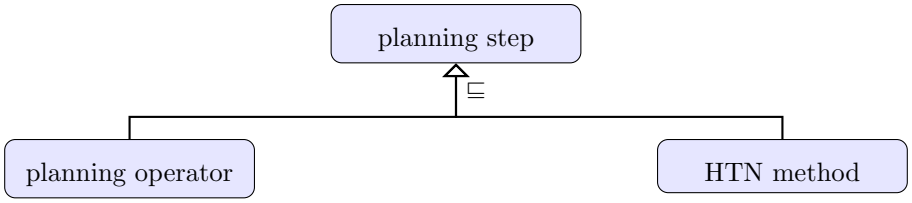



Figure 4.4.: Planning steps are an abstraction of planning operators and HTN methods.

be found. The application of a planning step maps the state of a planning process to a resulting state. This state is called the *planning state*. A planning state encodes all information that is necessary for the planning process. It is defined as follows:

**Definition 4.9** (planning state)**.** A *planning state* is a triple

$$ps := (ps_D, ps_p, ps_{kas})$$

in which the elements are described as follows:

- $ps_D$ is the current domain model.
- $ps_p$ represents the current plan.
- $ps_{kas}$ is a knowledge acquisition scheme.

A planning state $ps$ is called a *final* planning state if $ps_p$ is final and called an *intermediate* planning state if $ps_p$ is an intermediate plan. The set of planning states is denoted by $\mathcal{PS}$.

For a planning state $ps$, $ps_D$ is the current domain model of the planner. Additionally, a planning state can contain a knowledge acquisition scheme. This indicates a situation where the planner prefers to perform a knowledge acquisition process prior to continuing the overall planning process. How this works in detail is not important at this point and described in Chapter 5.

In the following, $\mathcal{P}$ denotes the set of plans, $\mathcal{KAS}$ the set of knowledge acquisition schemes, and $\mathcal{DM}$ the set of domain models.

A planning step is defined as follows:

**Definition 4.10** (planning step)**.** A *planning step* $\mathfrak{s}$ is represented by a 5-tuple

$$\mathfrak{s} := (\mathfrak{s}_{task}, \mathfrak{s}_{cond}, \mathfrak{s}_{\Delta plan}, \mathfrak{s}_{\Delta state}, \mathfrak{s}_{cost})$$

where

- $\mathfrak{s}_{task}$ is a task,

- $\mathfrak{s}_{cond}$ is a statement,
- $\mathfrak{s}_{\Delta plan}$ is a function $\mathfrak{s}_{\Delta plan} : \mathcal{P} \rightarrow \mathcal{P}$,
- $\mathfrak{s}_{\Delta state}$ is a function $\mathfrak{s}_{\Delta state} : \mathcal{DM} \rightarrow \mathcal{DM}$,
- and $\mathfrak{s}_{cost}$ is an ACog expression.

A planning step is an abstraction of an HTN method or a planning operator. In other words, any planning operator and HTN method constitutes a planning step. For a planning step $\mathfrak{s}$, the task $\mathfrak{s}_{task}$ defines—like for an HTN method—for which task a planning step is relevant. The precondition $\mathfrak{s}_{cond}$ and the cost $\mathfrak{s}_{cost}$ have the same function like for operators and methods. Hence, $\mathfrak{s}_{cond}$ describes the precondition that needs to be fulfilled in order to apply the planning step $\mathfrak{s}$, and $\mathfrak{s}_{\mathfrak{cost}}$ represents the expected cost of performing a task according to $\mathfrak{s}$.

In a nutshell, applying a planning step to a planning state has the following two consequences:

1. It transforms (i.e., decomposes) a plan into a resulting plan.
2. It transforms a domain model into a resulting domain model.

The first consequence is defined by the function $\mathfrak{s}_{\Delta plan}$, and the second consequence is defined by the function $\mathfrak{s}_{\Delta state}$. The functions $\mathfrak{s}_{\Delta plan}$ and $\mathfrak{s}_{\Delta state}$ together describe how a planning step maps a planning state to a resulting planning state.

The set $del(e_{del}, F)$ is the set that results from removing all members of $F$ that are unifiable with a member of $e_{del}$. For a planning step $\mathfrak{s}$ that represents a planning operator or planning method $e$, the function $\mathfrak{s}_{\Delta state}$ is defined as follows:

$$\mathfrak{s}_{\Delta state}(((F, C, R_D, R_G), act_M)) := ((del(e_{del}, F) \cup e_{add}, C, R_D, R_G), act_M) \quad (4.2)$$

The function $\mathfrak{s}_{\Delta state}$ updates the factual knowledge of a domain model such that all literals of the delete-set are removed, and all literals of the add-set are added to the set of facts.

The function $\mathfrak{s}_{\Delta plan}$ describes how a task is decomposed into a sequence of subtasks. If $\mathfrak{s}$ is a planning operator $o$ that is derivable with respect to a substitution $\sigma$, then the *plan transformation function* $\mathfrak{s}_{\Delta^o plan}$ is defined as follows:

$$\mathfrak{s}_{\Delta^o plan}((<t_1, \ldots, t_m>, <a_1, \ldots, a_n>)) := (<t_2, \ldots, t_m>, <a_1, \ldots, a_n, t_1\sigma>) \quad (4.3)$$

In other words, an instance of the next task is added to the end of the sequence of actions. The name of this task instance serves a symbolic representation of a corresponding agent control program that is executed during the execution phase.

If the planning step $\mathfrak{s}$ is a method $m$ that is relevant for a task $t_1$ and

$$m_{tasks} =< st_1, \ldots, st_m >,$$

then the plan transformation function $\mathfrak{s}_{\Delta^m plan}$ is defined as follows:

$$\mathfrak{s}_{\Delta^m plan}(< t_1, \ldots, t_n >, actions) := (< st_1, \ldots, st_m, t_2, \ldots, t_n >, actions) \quad (4.4)$$

Thus, the first task of the task sequence is replaced by the subtasks of the method. Based on Equation 4.3 and Equation 4.4, the plan transformation function $\mathfrak{s}_{\Delta plan}$ of a planning step $\mathfrak{s}$ is defined as follows:

$$\mathfrak{s}_{\Delta plan}(t, p) := \begin{cases} \mathfrak{s}_{\Delta^o plan}(t, p) & \text{if } \mathfrak{s} \text{ is a planning operator} \\ \mathfrak{s}_{\Delta^m plan}(t, p) & \text{if } \mathfrak{s} \text{ is an HTN method} \end{cases} \quad (4.5)$$

A planning step maps the current planning state to a resulting planning state. In this sense, operators map the current planning state to a resulting state by removing the next task from the task sequence, adding a ground instance of this task to the action sequence and updating the domain model according to the effects of the operator; whereas HTN methods transform the current planning state by replacing an active task by a number of subtasks, and updating the domain model according to its (high-level) effects. Applying a substitution $\sigma$ to a planning step $\mathfrak{s} := (\mathfrak{s}_{task}, \mathfrak{s}_{cond}, \mathfrak{s}_{\Delta plan}, \mathfrak{s}_{\Delta state}, \mathfrak{s}_{cost})$ is defined as applying $\sigma$ to every element of the tuple. Hence, $\mathfrak{s}\sigma$ is defined as follows:

$$\mathfrak{s}\sigma := (\mathfrak{s}_{task}\sigma, \mathfrak{s}_{cond}\sigma, \mathfrak{s}_{\Delta plan}\sigma, \mathfrak{s}_{\Delta state}\sigma, \mathfrak{s}_{cost}\sigma) \quad (4.6)$$

## 4.5. Execution Memory

The *execution memory* stores information that the agent gathers during the execution phase. For the proposed plan-based control architecture, one important type of information is the outcome of the execution of a knowledge acquisition task. This information is relevant in situations where the execution of a knowledge acquisition task cannot provide additional information. For example, consider a situation where a robot unsuccessfully tried to find Bob's mug by means of human robot interaction with Bob. Thus, Bob does not know where his mug is. Technically speaking, that means that the robot performed the following task:

$$\texttt{det}(\texttt{bob}, \texttt{location}(\texttt{bobs\_mug}, \texttt{Loc}), \varnothing, \varnothing, \texttt{indeterminable})$$

As specified by Definition 4.7, this constitutes a situation where the execution of the knowledge acquisition task could not provide additional information about the location of Bob's mug. However, the robot knows that Bob does not know the location of his mug. Thus, it is not reasonable—at least for a while—to submit the same query to Bob again. Therefore, the domain model stores information of the

performed knowledge acquisition tasks so that the planner can rule out knowledge sources that could not provide information about a certain query. For the illustrated example, the knowledge source Bob is called *unaware* with respect to the location of his mug.

More precisely, an unaware knowledge source is defined as follows:

**Definition 4.11** (unaware)**.** Let $act_{mem}$ be the execution memory of an activities model $act$. A knowledge source $ks$ is called *unaware* with respect to a knowledge acquisition task $\texttt{det}(ks, l, \texttt{C}, \texttt{I}, \texttt{X})$ iff $act_{mem}$ contains a knowledge acquisition task $\texttt{det}(ks, l', \texttt{C}', \texttt{I}', \texttt{indeterminable})$ such that

- $l$ is an instance of $l'$,
- $\texttt{I}'$ is a subset of $\texttt{I}$,
- and $\texttt{C}'$ is a subset of $\texttt{C}$.

Roughly speaking, Definition 4.11 defines that an external knowledge source cannot provide additional information with respect to a knowledge acquisition task if it failed to provide information with respect to a more general (i.e., a knowledge acquisition task that has less constraints) knowledge acquisition task. For example, if Bob cannot tell us the location of any mug, then it is assumed that he is also not able to give information about the location of a certain mug (e.g., $\texttt{mug77}$).

How relevant the information that is stored in the memory is, can dependent on when this information has been stored in the memory system. Often, older information is less relevant than more recently gathered information. Thus, it can be reasonable to "forget" outdated information or continuously decrease the relevance of stored information over time. However, the implementation of such a—more advanced—memory system is out of the scope of this work. In this work, it is assumed that all information in the memory system is relevant. Nevertheless, the proposed system does not rely on the fact that no information can be removed from the memory system. Hence, a more sophisticated memory system can be integrated without changing the proposed plan-based control system.

## 4.6. Definition of the Activities Model

All components have been introduced that are necessary in order to define the activities model. Based on the notion of planning operators, primitive actions and the execution memory, the activities model is defined as follows:

**Definition 4.12** (Activities Model)**.** An activities model is a triple

$$act = (act_o, act_m, act_{mem})$$

whereby the components are defined as follows:

- $act_o$ is a set of planning operators.
- $act_m$ is a set of HTN methods.
- $act_{mem}$ is a set of executed knowledge acquisition tasks.

## 4.7. Applicability

A knowledge acquisition scheme is only helpful for an agent if it is actually able to perform the corresponding knowledge acquisition tasks. For example, if a robot in principle is not able to find out whether a door is open, then the planner does not have to consider the method instances 2 and 3 for the situation illustrated by Figure 2.4. A knowledge acquisition scheme for which all necessary knowledge acquisition tasks can be possibly performed by the agent is called *possibly-acquirable* and more formally defined as follows:

**Definition 4.13** (possibly-acquirable)**.** An acquisition $(l, ks)$ is called *possibly-acquirable* with respect to a domain model $D_M$ iff there is a possibly-applicable planning step for a knowledge acquisition task $det(ks, l\sigma, I, C, l_r)$ such that $ks$ is not unaware with respect to $det(ks, l\sigma, I, C, l_r)$. Moreover, a knowledge acquisition scheme $kas$ is called possibly-acquirable iff $kas$ is empty or all $(l, ks) \in kas$ are possibly-acquirable.

In other words, an agent knows that information about certain aspects of the world can be acquired if it is in general able to generate a plan for a corresponding knowledge acquisition task, and the chosen knowledge acquisition task can (possibly) provide the required information. Knowledge acquisitions from sources that are believed to be unaware with respect to the corresponding knowledge acquisition task are ruled out.

Based on the notation of a possibly-acquirable knowledge acquisition scheme, it is possible to more precisely define the concept of a *possibly-applicable* planning step, which was informally introduced in Section 2.2. It is defined as follows:

**Definition 4.14** (possibly-applicable)**.** The instance $\mathfrak{s}\sigma$ of a planning step $\mathfrak{s}$ is called *possibly-applicable* with respect to a domain model $D_M$ and a knowledge acquisition scheme $kas$ iff $kas$ is possibly-acquirable and a knowledge acquisition scheme for $\mathfrak{s}_{cond}\sigma$.

Please note that if a planning step is possibly-applicable with respect to an empty knowledge acquisition scheme, then it is also applicable. A possibly-applicable planning step can only be applied after necessary information has been acquired by the execution of corresponding knowledge acquisition tasks. For example, consider the second method of the situation illustrated by Figure 2.4. This method instance can only be applied if the robot has perceived that `door2` is open. The fact that possibly-applicable planning step instances require the execution of additional tasks

(i.e., knowledge acquisition tasks) needs to be considered by the determination of the expected cost. Therefore, the cost of a possibly-applicable planning step is defined as the sum of the cost for the step if it is applicable and the expected cost of all necessary knowledge acquisition tasks.

## 4.8. Integrating Probabilistic Information

Probabilistic information about the expected outcome of a knowledge acquisition process can be integrated into the cost model such that the planner can minimize the overall expected cost by avoiding the execution of knowledge acquisition tasks that probably do not imply additional ways to perform the task at hand. In other words, the planner can trade off between plans that are more likely accomplishable and plans that have a lower expected cost. Let $l$ be a literal, *st* be a *statement* and $\mathfrak{p} \in [0, 1]$. How likely it is that there is a new instance of a literal $l$ can be defined by definite clauses of the following form:

$$p(l, \mathfrak{p}) \leftarrow st$$

For example, the definite clause

$$p(\texttt{open(X)}, 0.7) \leftarrow \top$$

defines that if a literal $l$ is unifiable with `open(X)` and open-ended, then the probability that there is an additional instance of $l$ is 0.7. Roughly speaking, that means that doors with an unknown state are assumed to be open in 70 percent of the cases. However, please note that these probabilities are only defined for open-ended literals. For example, if `door1` is known to be open, then `open(door1)` is not open-ended and the probability that door1 is open is 1. In other words, for literals that are not open-ended, the probabilistic information is not relevant. If no information about the probability of the existence of an additional instance of an open-ended literal is available, then a default value (e.g., 0.5) is used.

Let $\mathfrak{s}$ be a planning step that has the expected cost $c$ and is possibly-applicable with respect to a knowledge acquisition scheme *kas*. Moreover, let $\bigcup_i \{t_i\}$ be a set of knowledge acquisition tasks that contains a corresponding knowledge acquisition task for every knowledge acquisition $ka \in kas$, $c_i$ be the expected cost of $t_i$, and $E(t_i)$ be the event of finding the desired information of $t_i$. For a set of events $\bigcup_i E_i$, $p(\bigwedge_i E_i)$ denotes the probability that all events $E_i$ occur. The overall cost of $\mathfrak{s}$ is defined as follows:

$$\tilde{c}(\mathfrak{s}) := \frac{c + \sum_i c_i}{p(\bigwedge_i E(t_i))} \tag{4.7}$$

For example, let us reconsider the running example described in Section 2.1.2. Let us assume that the cost of the plan that results from applying the method for

`move_to(Room)` is always 100. Moreover, let us assume that the cost of performing the task

$$\texttt{det(laser,open(door2)},\varnothing,\varnothing,\texttt{l}_\texttt{r})$$

is 50, and the expected cost of performing the task

$$\texttt{det(percept,connect(lab,X,kitchen)}, \{\texttt{connect(lab,door1,kitchen)}, \texttt{connect(lab,door2,kitchen)}\}, \{\texttt{open(X)}\},\texttt{l}_\texttt{r})$$

is 300. Furthermore, we assume that the probability that a door with an unknown state is open is 0.7, the probability of finding an additional connection between the kitchen and the lab is expected to be 0.1, and both events are statistically independent.

In this situation, the cost of method instance 1 is 100, the cost of method instance 2 is $\frac{100+50}{0.7} = 214$, and the cost of method instance 3 is $\frac{100+50+300}{0.7\times0.1} = 6428$. Thus, in this case, the applicable instance has the less expected cost. However, this does not always has to be the case. For example, if the robot is directly located in front of door2, and the expected cost of moving to the kitchen via door2 is only 10, then the overall cost of method instance 2 is $\frac{10+50}{0.7} = 86$. Thus, in this case, the planner chooses method instance 2 and decides to determine whether door2 is open prior to continuing the overall planning process.

In terms of *utility theory* (Russell and Norvig, 2010), the expected cost $c$ can also be viewed as a *utility* $\frac{1}{c}$. The calculation of the cost $\tilde{c}$, defined by Equation 4.7, can be transformed into the following utility function:

$$\tilde{u}(\mathfrak{s}) := \frac{p(\bigwedge_i E(t_i))}{c + \sum_i c_i} \tag{4.8}$$

Based on that representation, the proposed domain model enables the planner to follow the principle of *maximum expected utility* (Russell and Norvig, 2010, page 611) by means of choosing the planning step that maximizes the agent's utility.

The proposed domain model does not provide advanced probabilistic reasoning capabilities. However, external probabilistic reasoners can be integrated in various ways. For example, they can be called via using the **call** meta-predicate (see Section 3.7.2).

## 4.9. High-Level Percepts: Defining Multimodal Integration Processes

Robots are usually equipped with different sensing and acting modalities. The integrative processing of different sensing and acting modalities is an essential approach to provide reliable information about complex, dynamic environments (Luo and

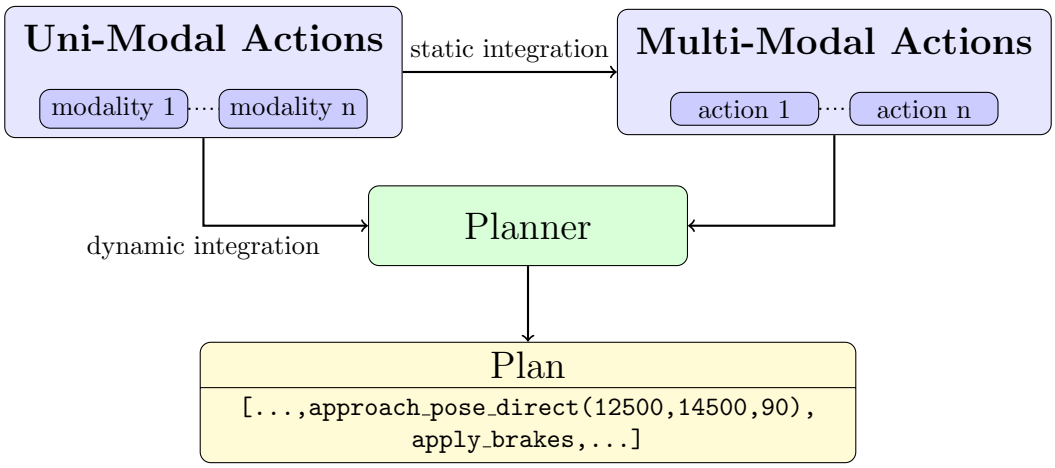



Figure 4.5.: Illustration of multi- and unimodal integration processes.

Kay, 1995). The plan-based control architecture can integrate sensing and acting modalities in the following—not exclusive—two ways (Off and Zhang, 2011b):

1. Acting and sensing modalities are composed to multimodal primitive actions.
2. Each sensing and acting modality is an unimodal robot action and dynamically integrated by symbolic planning to reasonable multimodal behavior.

These two integration approaches are illustrated by Figure 4.5. The former approach is usually achieved by means of implementing a specific control program that achieves the desired multimodal integration. The advantage of this approach is that different modalities can be more tightly integrated than with the second approach. Nevertheless, the main disadvantage is that the integration is rather static and out of the control of the plan-based controller.

The second approach, however, supports the dynamic integration of different sensing and acting modalities. Actions that are using a single modality can dynamically be integrated by the planner to a multimodal sequence of actions. Multimodal integration processes are defined in form of *high-level percepts* in order to better support this approach. For the purpose supporting high-level percepts, HTN methods, as defined by Definition 4.4, are extended to the following 7-tuple:

$$m := (m_{task}, m_{cond}, m_{del}, m_{add}, m_{tasks}, m_{percept}, m_{cost})$$

This 7-tuple contains the additional term $m_{percept}$. The term $m_{percept}$ is an ACog expression (see Definition 4.1) that defines how a percept is determined based on the percepts of subtasks.

For example, Figure 4.6 shows a method specification for the task that determines whether the path between two waypoints is free via using the laser scanner. The definition is based on the primitive sensing action `free_ahead(D)` that can determine how many millimeters a path is free straight ahead of the robot. Based on that, the

```
method(
  % task
  det(laser,free_path(W1,W2),I,C,R),
  % precondition
  (at(robot,W1), dist(W1,W2,D1)),
  % delete-set
  [],
  % add-set
  [],
  % subtasks
  [rotate_towards(W2), sense(free_ahead(D2),R1)],
  % high-level percept
  eval(R,call((
  (
    D1 < D2 ->
    R = free_path(W1,W2)
    ;
    R = impossible))),
  % cost
  200).
```

Figure 4.6.: Example HTN method for an acquisition task with a high-level percept.

high-level percept defines that the path between two waypoints `W1` and `W2` is free if the robot is at `W1`, rotates towards `W2`, and the path that is then free ahead of the robot is bigger than the distance between the waypoints.

## 4.10. Discussion and Related Work

This chapter has described the activities model of the proposed plan-based control system. The activities model encapsulates knowledge about the activities that can be performed by the corresponding agent. The activities model and the state model, described in Section 3, constitute the overall domain model.

The proposed activities model is most closely related to the activities models of SHOP (Nau et al., 1999) and SHOP2 (Nau et al., 2003). Compared to the activities model of SHOP and SHOP2, several extensions have been proposed in order to deal with an open-ended state and support the reasoning of active knowledge acquisition. Similar to the activities models of other HTN planners including NONLIN (Tate, 1977), SIPE (Wilkins, 1983), O-Plan (Drummond and Currie, 1989), SIPE-2 (Wilkins, 1990), UMCP (Erol et al., 1994), DPOCL (Young et al., 1994), AbNLP (Fox and Long, 1995), the proposed model encodes hierarchical knowledge that is used by the planner in order to more efficiently solve planning problems. High-level effects are also supported by the work of (Marthi et al., 2007, 2008).

In contrast to existing HTN planning approaches, the proposed domain model is not based on the closed world assumption. According to the requirements formulated in Section 2.3, one of the objective of this work is to develop an HTN planner that is able to reason about relevant, possible, and acquirable domain model extensions. For the purpose of achieving this goal, the task of the proposed domain model is to provide the basic knowledge representation and reasoning techniques for such a planner. The activities model described in this chapter extends existing approaches in several ways in order to provide these techniques.

One of the main new idea of the proposed activities model is the notion of a knowledge acquisition task. Knowledge acquisition tasks are defined using a syntactical convention. They represent tasks that try to determine a new instance of a literal from a certain external knowledge source. Like for any other task, HTN methods can describe how a knowledge acquisition task can be decomposed into a sequence of subtasks. Based on the definition of HTN methods for knowledge acquisition tasks, the domain model can determine (1) what knowledge acquisitions are possible under what conditions, (2) how expensive it is to acquire information from a specific knowledge source, and (3) how a knowledge acquisition task can be performed.

Another important contribution of the proposed domain model is the extension of the notion of an applicable HTN method. The definition of an applicable HTN method (Ghallab et al., 2004, Definition 11.3) is extended to the notion of a possibly-applicable planning step (see Definition 4.14). The notion of a possibly-applicable planning step considers that a planning step can be applicable, though the precondition of the applicability depends on information that cannot be derived from the domain model at hand. In this way, the domain model points out additional ways to continue the planning process that require additional information. Thus, it points out more ways to perform a given task. This is particularly relevant in situations where it would otherwise be impossible to find any plan at all, or all other plans are suboptimal.

Moreover, existing domain models are extended by means of providing a simple execution memory that stores information about executed knowledge acquisition tasks for the purpose of avoiding it to submit queries to external knowledge sources that are known to be unable to provide the desired information.

Another new feature is the concept of high-level percepts. They can be defined as part of an HTN method. High-level percepts describe how a high-level percept can be determined based on the outcome of the execution of subtasks. In this way, high-level percepts can describe multimodal integration processes so that a planner can automatically combine a set of primitive actions to a desired multimodal behavior.

In principle, there are two ways to create an instance of a domain model:

1. The domain model is created by a human domain expert.

2. The domain model is the result of a (automatic) learning process.

Both approaches can be combined so that some parts are specified by a human expert and others are learned automatically. For the plan-based controller, which exploits the domain model, it is of no consequence how the domain model has been created. The domain models used in this work are all created by a human domain expert. However, if a learning approach is available that automatically creates domain models, then the automatic learning process can be used without changing the proposed plan-based control system.

The domain model and its reasoning services are integrated into the overall plan-based control system. The following chapter describes how the planner uses the proposed domain model.

Chapter 5

# HTN Planning in Open-Ended Domains

*Research on planning for robots is in such a state of flux that there is disagreement about what planning is and whether it is necessary.* (McDermott, 1992)

## Contents



The core component of the plan-based robot control architecture is a new HTN planning system. This planning system is called *ACogPlan* (Artificial Cognitive Systems Planner). The general idea of ACogPlan has already been illustrated in Section 2.2. The idea is to additionally consider ways to continue the planning process that require the acquisition of additional information. The objective of this chapter is to describe how this idea can actually be implemented.

ACogPlan is based on the open-ended domain model described in Chapter 3 and 4. In contrast to existing HTN planning systems, the planning process does not have to be monolithic. In other words, the planner is not based on the assumption that a complete plan can be generated prior to executing any action. The plan-based control system constitutes a continual planning system (Brenner and Nebel, 2009) that interleaves planning and execution so that missing information can be acquired by

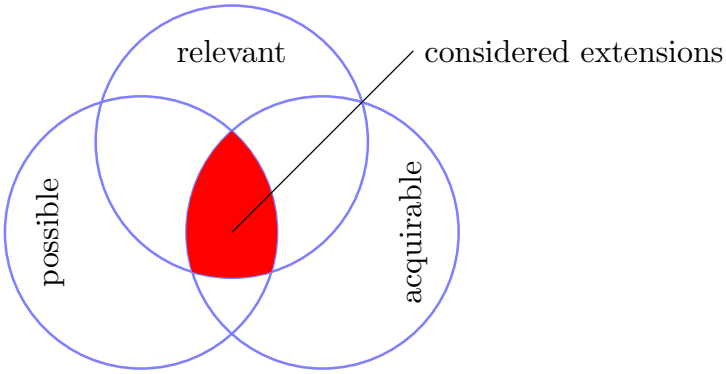



Figure 5.1.: Illustration of the fact that the planner only considers extensions that are *relevant*, *possible*, and *acquirable* with respect to a statement.

means of active knowledge acquisition. The overall planning and execution process usually consists of several planning and execution phases. ACogPlan is responsible for the planning phases. It generates possibly incomplete plans and automatically decides when it is more reasonable to acquire what additional information prior to continuing the overall planning and execution process. In this way, the planner decides that a corresponding knowledge acquisition task is executed before it continues the planning process. Generally speaking, it automatically decides when the system should switch between planning and execution.

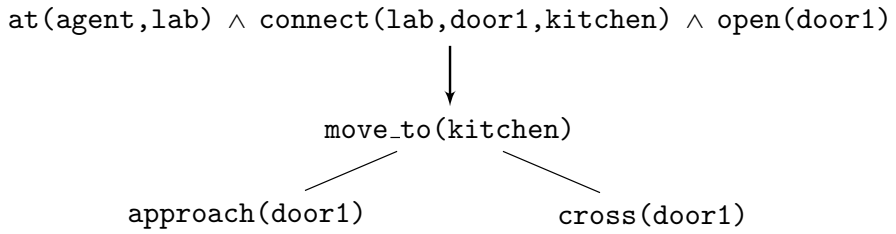
The proposed planner extends existing HTN planners in several ways for the purpose of implementing the aforementioned strategies. The core algorithm of ACogPlan can be seen as an extension of the algorithm of the SHOP (Nau et al., 1999) planner. A preliminary version of ACogPlan has been presented in Off and Zhang (2012).

## 5.1. Requirements

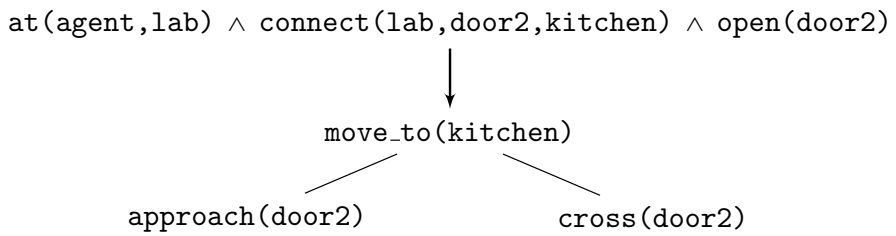
The general idea of the proposed planner is to also consider HTN method or planning operator instances that are only possibly-applicable. In order to do that, a planner must be able to reason about extensions of a given domain model. However, a planner should not consider all extensions of a domain model at hand. As enumerated in Section 2.3, a considered extension should be *relevant*, *possible*, and *acquirable*. This section describes how a planner can use the domain model described in Chapter 3 and 4 in order to figure out which extensions fulfill these constraints.

### 5.1.1. Relevant Domain Model Extensions

For the purpose of enabling a planner to only consider relevant extensions of the domain model at hand, we first have to answer the following question: What is a relevant extension? More precisely, we have to find an answer to the following




(a) The shown instance of the precondition enables it to decompose the task of moving into the kitchen into the task of approach and cross `door1`




(b) The shown instance of the precondition enables it to decompose the task of moving into the kitchen into the task of approach and cross `door2`

Figure 5.2.: This Figure shows two examples that illustrate that each additional instance of a precondition implies an additional way to continue the planning process.

question: What additional information implies that the planning process can be continued in an alternative manner? Based on that question, the *relevant extension* of a domain model is defined as follows:

**Definition 5.1** (relevant extension)**.** A domain model extension is called *relevant* with respect to a planning state iff it implies that the planning process can be continued in an alternative manner.

However, now the question is: What domain model extensions imply that the planning process can be continued in an alternative manner? For the purpose of answering this question we have to take a look at the general idea of an HTN planning algorithm—as described in Section 2.1—again. HTN planning proceeds via iteratively applying relevant HTN methods and planning operators such that an abstract plan is refined into a sequence of primitive tasks that can be executed by a corresponding agent. An HTN method or a planning operator can only be applied if its precondition holds. If we take a look at this aspect from another perspective, then we can observe that every additional instance of a precondition implies an additional instance of the corresponding planning step. This is exemplified by Figure 5.2. Figure 5.2 shows which ways (e.g., decompositions) to continue the planning process are implied by an instance of the precondition of the second method shown in Figure 2.2 (i.e., the method for the task `move_to(R)`). Figure 5.2a shows a precondition that enables the planner to decompose the task `move_to(kitchen)` into

the task of approaching and crossing `door1`, whereas Figure 5.2b shows an instance of the precondition which implies that the task can be decomposed into the task of approaching and crossing `door2`. Based on this observation, the following can be proposed:

**Proposition 5.1.** *A domain model extension is* relevant *iff it implies that an additional instance of the precondition of a relevant HTN method or planning operator can be derived.*

*Proof.* It directly follows from the definition of the HTN planning algorithm (see Algorithm 5.1) that every additional instance of the precondition of a relevant HTN method or planning operator implies an additional way to continue the planning process. □

Based on Proposition 5.1, we know that a planner needs to look for information that implies that an additional instance of a relevant precondition can be derived, since that implies that the planning process can be continued in an alternative manner. In order to determine which information implies an additional instance of a precondition, the planner can use the domain model ACogDM and the reasoning system ACogReason described in Chapter 3 and 4. More precisely, the planner can ask the reasoner whether a relevant precondition is possibly-derivable, since according to the definition of a possibly-derivable statement (see Definition 3.8), a new instance of all open-ended literals implies an additional instance of the overall statement.

For example, let us consider the situation described in Section 2.1.2 again. For the purpose of finding a way to decompose the task `move_to(kitchen)` into a sequence of subtasks, the planner can submit the precondition of the relevant method (see Figure 2.2) to the reasoning system. As illustrated by Figure 5.3a - 5.3c, the reasoner ACogReason returns three solutions. Each solution tells the planner that the precondition is possibly-derivable with respect to a set of open-ended literals. Roughly speaking, the solutions tell the planner that:

1. it is possible to move to the kitchen via `door1` without additional knowledge acquisition (see Figure 5.3a),

2. it is possible to move to the kitchen via `door2` if the corresponding agent can find out that `door2` is open (see Figure 5.3b),

3. and it is possible to move to the kitchen via an unknown door `X` if the corresponding agent can find a new door `X` which connects the lab with the kitchen and is open (see Figure 5.3c).

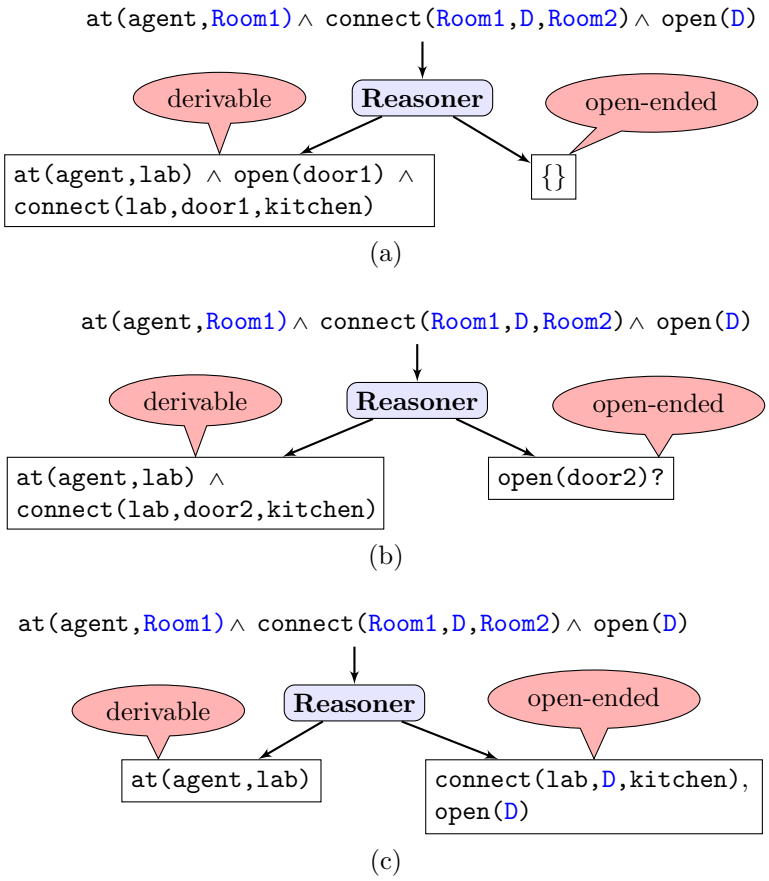



Figure 5.3.: All possible results of the reasoner for the situation illustrated by Figure 2.4.

### 5.1.2. Possible Extensions

Although an agent will probably never have a perfect domain model of a real-world environment, the planner is supposed to exploit all information (e.g., subconcept relations or knowledge about the maximum number of instances) it has in order to detect and rule out a large set of extensions that are impossible with respect to the current state of the world. The state model described in Chapter 3 provides the required knowledge representations and reasoning services for this objective. Therefore, the planner that calls reasoning services of the state model as a subroutine does not have to deal with these issues directly.

### 5.1.3. Acquirable Extensions

An agent is only allowed to perform—possibly partial—plans if they are the result of the successive application of relevant and applicable planning steps. Possibly-applicable planning steps constitute additional ways to continue the planning process, but cannot directly be applied in order to ensure a relaxed form of soundness,

namely *prefix-soundness*. The notion of a *prefix-sound* plan is described in more detail in Section 5.4. Roughly speaking, an intermediate prefix of a plan is called prefix-sound if it is likely that it constitutes the prefix of a sound and complete plan. In order to ensure the prefix-soundness property, a possibly-applicable planning step can only be applied after additional information has been acquired that implies that it is applicable. However, it is only possible to determine that information if the agent is able to perform the necessary knowledge acquisition tasks. Otherwise, the planner does not have to consider related ways to continue the planning process.

Based on the specification of knowledge acquisition tasks (see Section 4.3), the reasoner ACogReason is able to determine which knowledge acquisition processes are possible. Such a knowledge acquisition is called *possibly-acquirable* (see Definition 4.13). Based on Definition 4.13, the reasoner can tell the planner which extensions of a domain model are possibly acquirable. Hence, this information can be provided by the domain model. The only thing that needs to be done by the planner is to call the appropriate reasoning process.

## 5.2. Core Planning Algorithm

The core algorithm of the proposed HTN planning system is shown by Algorithm 5.1. It describes the planning phase of the continual planning and acting approach that usually consists of several planning phases. The algorithm can be seen as an extension of the SHOP (Nau et al., 1999) algorithm that additionally considers the possibly-applicable HTN method and planning operator instances.

A planning state (see Definition 4.9) is the input and the output of the recursive planning algorithm. If the planning state is final, then the planning process has successfully generated a complete plan and the given planning state is returned (lines 1-2). Otherwise, the algorithm chooses the possibly-applicable step with the lowest expected cost. The cost is calculated using probabilistic information as defined by Equation 4.7. Hence, the planner takes the additional cost of possibly necessary knowledge acquisition tasks and the probability of considered domain model extensions into account for the purpose of making a decision with the lowest expected cost. If we view the expected cost $c$ as a utility $\frac{1}{c}$ and transform the Equation 4.7 into the utility function defined by Equation 4.8, then the planner follows the principle of *maximum expected utility* (Russell and Norvig, 2010, page 611). Thus, the planner chooses the action that maximizes the expected utility.

If the planner chooses an applicable planning step (i.e., no knowledge acquisition is necessary and the knowledge acquisition scheme is the empty set), then planning proceeds similar to the SHOP (Nau et al., 1999) planner. The only additional factors that needs to be considered are *high-level effects*. The effects of HTN methods are only applied after all subtasks have been performed. Therefore, the planner creates a plan that only contains the next task (line 7), transforms this plan according to the definition of the planning step (line 8), and continues the planning process only for the next task. If the planner has generated a complete plan for the next task, then

**Algorithm 5.1:** plan($ps$)

**Result**: a planning state $ps'$, or failure

1 **if** *ps is a final planning state* **then**
2  **return** $ps$;
3 $steps \leftarrow \{(\mathfrak{s}\sigma, kas) | \mathfrak{s}$ is the instance of a relevant planning step, $\sigma$ is a substitution such that $\mathfrak{s}\sigma$ is relevant for the next task, and $\mathfrak{s}\sigma$ is possibly-applicable with respect to $ps_D$ and $kas\}$;
4 **if** `backtrackable-choose` $(\mathfrak{s}\sigma, kas) \in$ *steps with the minimum overall cost* **then**
5  **if** $kas = \varnothing$ **then**
6  Let $(ps_p)_{tasks} =< t_1, \ldots, t_n >$;
7  $p' \leftarrow (< t_1 >, (ps_p)_{actions})$;
8  $p'' \leftarrow \mathfrak{s}_{\Delta plan}(p')$;
  `/* generate a plan for the next task */`
9  $ps' \leftarrow$ `plan`$((ps_D, p'', ps_{kas}))$;
10  **if** $ps'$ *is final* **then**
11  $D'_M \leftarrow \mathfrak{s}_{\Delta state}(ps'_D)$;
  `/* generate a plan for the remaining tasks */`
12  `plan`$((D'_M, (< t_2, \ldots, t_n >, (ps'_p)_{actions}), ps'_{kas}))$;
13  **else if** $ps'$ *is an intermediate planning state* **then**
14  **return** $ps'$;
15  **else**
16  **return** $(ps_D, ps_p, kas)$;
17 **else**
18  **return** *failure*;

the effects are applied to the domain model (line 11), and the planner is recursively called in order to generate a plan for the remaining tasks. However, if the planner stops the planning process for the next task and returns an intermediate planning state, then the (superordinate) planner instance stops the planning process for the overall task and returns the intermediate planning state.

If the planner chooses an only possibly-applicable planning step at line 4, then it stops the planning process and returns the current (intermediate) planning state including the non-empty knowledge acquisition scheme of the chosen planning step (lines 15-16). This indicates a situation where the planner decides that it is more reasonable to acquire additional information prior to continuing the planning process of the overall task. By means of choosing an applicable or only possibly-applicable planning step, the planner decides whether it is more reasonable to continue the planning (i.e., choose an applicable planning step) or to first acquire additional information (i.e., choose an only possibly-applicable step). In this way, it automati-

```
operator(sense(laser,open(Door),I,C,R),
  % precondition
  (approached(Door)),
  % delete-set
  [],
  % add-set
  [],
  % expected cost
  500).
```

Figure 5.4.: Planning operator specification for the sensing action that determines the state of a door using the laser scanners.

cally decides when to switch between planning and acting. If it is neither possible to continue the planning process nor to acquire relevant information, then the planner backtracks to the previous choice point or returns *failure* if no such choice point exists.

## 5.3. Planning in Open-Ended Domains: An Example

This section is intended to illustrate the proposed planning system using a simple example. Let us consider an extended version of the example introduced in Section 2.1.2. In addition to the four primitive actions

$$\texttt{approach(Entity)}, \texttt{cross(Door)}, \texttt{pick\_up(Obj, Table)}, \texttt{put\_down(Obj, Table)}$$

the robotic agent can sense the state of doors using its laser scanner by means of the primitive action `sense(laser,open(Door),I,C,R)`. The parameters of this sensing action have the same meaning as for a knowledge acquisition task. Hence, `laser` is an external knowledge source, `open(Door)` the literal for which a new instance should be determined, `I` the set of derivable instances of `open(Door)`, `C` is the context, and `R` the result of the sensing action. For this sensing action, the planning operator specification shown in Figure 5.4 is added to the four planning operator specifications shown in Figure 2.1.

Furthermore, HTN method specifications shown in Figure 2.2 are extended. The resulting method specifications are shown in Figure 5.5.

Like in the example described in Section 2.1.2, the state model contains the following set of facts:

```
in_room(agent,lab), in_room(t1,lab), in_room(t2,kitchen),
on(bobs_mug,t1), free(hand), connect(lab,door1,kitchen),
connect(lab,door2,kitchen), open(door1).
```

```
method(deliver(Obj,Table),
  % precondition
  (on(Obj,Table1) ∧ in_room(Table,R) ∧
    can_navigate_to(Table,NavCost1) ∧
    can_navigate_to(Table1,NavCost2))
  % subtasks
  [approach(Table1),pick_up(Obj),
    move_to(R),approach(Table),put_down(Obj,Table)],
  % expected cost
  eval(Cost, Cost is NavCost1 + NavCost2 + 5000)).
method(move_to(R),
  % precondition
  (in_room(agent,R1) ∧ connect(R1,Door,R) ∧ open(Door) ∧
    can_navigate_to(Door,NavCost))
  % delete-set
  [in_room(R1)],
  % add-set
  [in_room(R)],
  % subtasks
  [approach(Door),cross(Door)],
  % expected cost
  eval(Cost,Cost is NavCost + 2000)).
method(determine(laser,open(Door),I,C,R),
  % precondition
  (in_room(agent,R1) ∧ connect(R1,Door,_) ∧
    can_navigate_to(Door,NavCost))
  % subtasks
  [approach(Door),sense(laser,open(Door),I,C,R)],
  % expected cost
  eval(Cost,Cost is NavCost + 500)).
```

Figure 5.5.: Methods for the illustrative pick-up and delivery task.

For the example at hand, we additionally assume that the following literals can be derived by domain specific rules of the state model:

```
can_navigate_to(door1,10000), can_navigate_to(door2,500),
can_navigate_to(t1,1500), can_navigate_to(t2,12000).
```

Hence, the robotic agent has a model of the expected cost for the task of navigating to `door1`, `door2`, `t1`, and `t2`. For example, the expected cost of navigating to `door1` is 10000, whereas the expected cost of navigating to table `t1` is 1500.

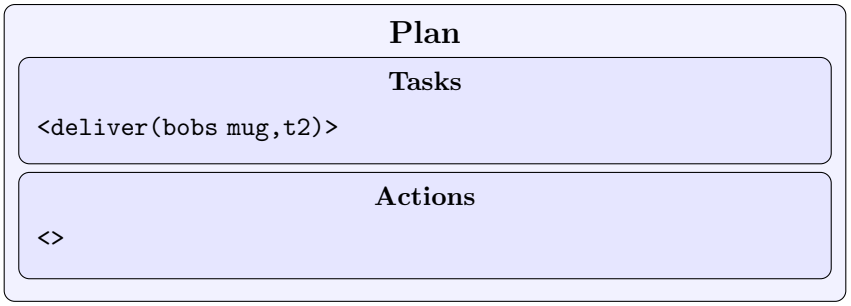



Figure 5.6.: Initial plan of the illustrated example.

In the following, it will be described how the planning algorithm deals with the input plan shown in Figure 5.6.

The planner starts the planning process by means of looking for planning steps that are relevant for the task `deliver(bobs_mug,t2)`. For the example at hand, the only relevant planning step is the first method shown in Figure 5.5. The planner instantiates the method via substituting `Obj` with `bobs_mug` and `Table` with `t2`. Subsequently, the planner calls the reasoner for the purpose of determining all possibly-applicable (see Definition 4.14) instances of the method. In this case, there is only one possibly-applicable instance with an empty knowledge acquisition scheme. A 2-tuple consisting of an instance of the method and a corresponding empty knowledge acquisition scheme is shown by Figure 5.7.

```
(
  method(deliver(bobs_mug,t2),
    (on(bobs_mug,t1) ∧ in_room(t2,kitchen) ∧
      can_navigate_to(t2,12000) ∧
      can_navigate_to(t1,1500))
    [],
    [],
    [approach(t1),pick_up(bobs_mug),
      move_to(kitchen),approach(t2),put_down(bobs_mug,t2)],
    eval(Cost, Cost is 12000 + 1500 + 5000)),
  {})
```

Figure 5.7.: All possibly-applicable instances of the relevant method for the task `deliver(bobs_mug,t2)`.

The expected cost of the method instance is 12000 + 1500 + 5000 = 18500. The planner chooses the described instance (see Algorithm 5.1, line 4), since this is the only instance of the set *steps* generated by Algorithm 5.1 (line 3). Due to the fact that the corresponding knowledge acquisition scheme is empty, the planning algorithm continues at line 6. It decomposes the task `deliver(bobs_mug,t2)` into the following sequence of subtasks (line 8):

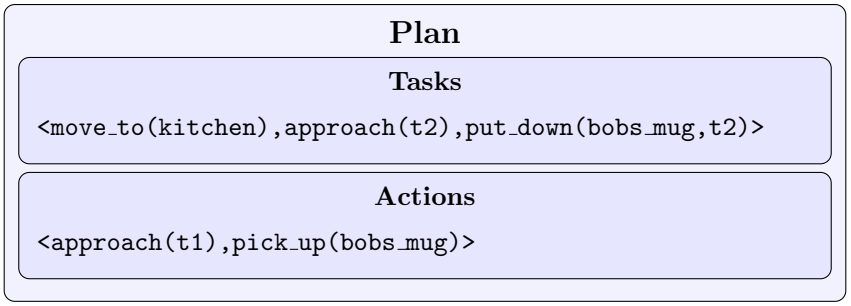



Figure 5.8.: Plan after the planner has applied an operator to the tasks `approach(t1)` and `pick_up(bobs_mug)`.

```
<approach(t1),pick_up(bobs_mug),move_to(kitchen),approach(t2),
   put_down(bobs_mug,t2)>
```

Afterwards, the planner recursively calls itself with the updated plan (line 9). Subsequently, the planner applies an operator to the task `approach(t1)` and to the task `pick_up(bobs_mug)`. For both primitive tasks, there is only one possibly-applicable planning operator available that requires no additional knowledge acquisition. Figure 5.8 shows how the plan of the planning state looks after these two planning operators have been applied.

Hence, the plan already contains a sequence of two executable actions. The next task for which a plan has to be generated is `move_to(kitchen)`. There are two possibly-applicable method instances for this task. These method instances and the corresponding knowledge acquisition scheme are shown as a 2-tuple in Figure 5.9. In this case, the planner does not consider the possibility that there is an unknown door which connects the lab with the kitchen, since the domain model contains no planning step that enables the planner to find new connections between rooms. Hence, a corresponding domain model extension is not acquirable. According to Equation 4.7, the cost of the first instance is $12000 + 2000 = 14000$. The cost of the second instance is $500 + 2000 = 2500$ plus the cost for the knowledge acquisition task. The planner instantiates the relevant method for the knowledge acquisition task that determines the state of a door (see Figure 5.5). The cost for the knowledge acquisition task is $500+500 = 1000$. The probability that the door is open is assumed to be 0.7. Thus, the overall cost of the second method is

$$\frac{2500 + 1000}{0.7} = 5000.$$

Hence, for the illustrated example the second method instance has the lowest expected cost. Therefore, the planner chooses the second instance. However, now the planner has chosen an only possibly-applicable planning step. Thus, it decided that it is more reasonable to determine the state of `door1` prior to continuing the overall planning and execution process. Finally, the planner returns a planning state that consists of the current domain model, the current plan, and the knowledge

```
(
  method(move_to(kitchen),
    (in_room(agent,lab) ∧ connect(lab,door1,kitchen) ∧
      open(door1) ∧ can_navigate_to(door1,12000))
    [in_room(lab)], [in_room(kitchen)],
    [approach(door1),cross(door1)],
    eval(Cost,Cost is 12000 + 2000)),
  {})

( method(move_to(kitchen),
    (in_room(agent,lab) ∧ connect(lab,door2,kitchen) ∧
      open(door2) ∧ can_navigate_to(door2,500))
    [in_room(lab)], [in_room(kitchen)],
    [approach(door1),cross(door1)],
    eval(Cost,Cost is 500 + 2000)),
  {det(laser,open(door2),{},{},R)})
```

Figure 5.9.: All possibly-applicable instances of the method for the task `move_to(kitchen)`.

acquisition scheme of the second method instance shown in Figure 5.9 (see Algorithm 5.1, line 16).

## 5.4. Soundness and Completeness

Two important properties of a planning system are the *soundness* and the *completeness*. A planning algorithm is called sound if all generated plans are correct and called complete if the algorithm finds a solution for any solvable problem.

Defining the soundness and completeness is more complicated for the proposed planning algorithm, since it is not assumed that the planner can generate a complete plan. However, if the planning algorithm generates a complete plan, then it behaves like—and generates the same plans as—the SHOP planning system. SHOP is known to be sound for all search spaces and complete for finite search spaces (Nau et al., 1999). Therefore, Algorithm 5.1 is also sound for all search spaces and complete for finite search spaces with respect to complete plans. Iterative deepening can be used to also make the algorithm complete for infinite search spaces. Nevertheless, according to Nau et al. (1999), in practice it turned out to be more efficient not to use iterative deepening. If the planner calls external reasoners, then the completeness and soundness are based on the assumption that the external reasoners are complete and sound.

The proposed planning system can also be seen as complete with respect to partial plan prefixes, since it generates all plan prefixes that can be the prefix of a sound plan. However, only generating a partial plan prefix comes at the cost of loosing the

guarantee that the generated plan prefix is the prefix of a sound plan. Nevertheless, if the planner only has an open-ended model of its environment, then the alternative to start execution prior to having a complete plan would often be not to perform a task at all. Furthermore, in a dynamic environment even sound plans can quickly become invalid if relevant aspects of the situation have been changed.

A partial plan prefix cannot be guaranteed to be sound, but a relaxed form of soundness can be guaranteed. Every generated plan prefix is *prefix-sound*, whereas prefix-sound is defined as follows:

**Definition 5.2** (prefix-sound)**.** A plan prefix is called *prefix-sound* if it is the intermediate plan prefix of a sound planning algorithm.

This definition seems to be very weak. How much does it actually tell us about the correctness of a plan? The answer to this question is: It depends on the concrete planning algorithm. The notion of prefix-sound is not reasonable for any planning approach. However, for the forward search HTN planner ACogPlan it is reasonable due to the following reasons:

1. Like SHOP (Nau et al., 1999), ACogPlan generates plans in the same order in which they will later be executed. Therefore, the sequence of actions of an intermediate plan constitutes the prefix of a sound plan if the planner can generate a complete plan without discarding these actions.

2. The fact that the search process is guided by HTN methods can significantly reduce necessary backtracking. How significant this effect is depends on the quality of the HTN methods. In other words, if the planner has a good domain model, then intermediate plans often constitute the prefix of a sound and complete plan.

3. If it is necessary to discard some of the actions from the intermediate sequence of actions, then the planning algorithm starts at the end of the sequence. Thus, even if some actions of an intermediate plan are discarded, a prefix of this prefix can still be the prefix of a complete, correct plan.

## 5.5. Discussion and Related Work

The new HTN planning system ACogPlan has been described in this chapter. The proposed planning system is particularly made for real-world situations where the planner only has an incomplete, open-ended model of the environment. It is based on the open-ended domain model described in Chapter 3 and 4.

Most of the previous approaches on automated planning that are able to generate plans in partially known environments generate conditional plans—or policies—for all possible contingencies. This includes *conformant*, *contingent* or *probabilistic* planning approaches (Russell and Norvig, 2010; Ghallab et al., 2004). Work on

generating conditional plans includes (Hoffmann and Brafman, 2006; Petrick and Bacchus, 2002, 2004; Weld et al., 1998; Levesque, 1996; To et al., 2011), to name but a few. The *PKS* planner (Petrick and Bacchus, 2002, 2004) has also been applied in a robotic context (Kraft et al., 2008). How a closed-world, hierarchical planner can be adapted so that it can generate conditional plans for incomplete domains is described in Kuter et al. (2007). However, planning approaches that generate conditional plans are known to be computationally hard (Littman et al., 1998; Rintanen, 1999; Baral et al., 2000; Rintanen, 2004), scale badly in open-ended domains, and are only applicable if it is possible to foresee all possible outcomes of a sensing action.

Several planning approaches that are able to deal with incomplete information use *runtime variables* to generate conditional plans or interleave planning and execution including Ambros-Ingerson and Steel (1988); Etzioni et al. (1992); Golden (1998); Knoblock (1995). Runtime variables can be used as action parameters and enable the reasoning about unknown future knowledge. Nevertheless, reasoning about runtime variables is heavily limited, since the only thing that is known about them is the fact that they have been sensed.

The most closely related previous work is Brenner and Nebel (2009). The proposed continual planning system also interleaves planning and execution so that missing information can be acquired by means of active knowledge acquisition. In contrast to our work, this approach is based on classical planning systems that do not natively support the representation of incomplete state models and are unable to exploit domain specific control knowledge in the form of HTN methods. Moreover, it is not stated whether the approach can deal with open-ended domains in which it is not only necessary to deal with incomplete information, but also essential to, for example, consider the existence of a priori completely unknown objects or relations between entities of a domain. Furthermore, the approach is based on the assumption that all information about the precondition of a sensing action is a priori available and thus will often (i.e., whenever this information is missing) fail to achieve a given goal in an open-ended domain. The approach has been used in the service robotic context (Keller et al., 2010), but there is no comprehensive study that gives more detailed information about the performance in a real-world robotic environment.

Another interesting continual planning approach proposed by Goebelbecker et al. (2011) combines a classical and a decision theoretic planner. However, the approach cannot deal with situations where a knowledge acquisition task requires the acquisition of additional information and thus would fail to deal with scenarios described in Chapter 7 and 9.

Furthermore, Talamadupula et al. (2010b,a) propose *Open World Quantified Goals (OWQG)*, that can be used in order to enable a closed-world planner to deal with goals about previously unknown objects. Compared to the proposed framework, the usage of OWQG provides a less autonomous behavior, since OWQG's encode hand-coded knowledge about possibly sensed object instances and related goals.

Hierarchical planning has been successively used in the robotic context (Hartanto,

2009; Wolfe et al., 2010; Weser et al., 2010). Furthermore, hierarchical planning has been extended to deal to some degree with incomplete information and more dynamic situations where the domain model is updated during execution. An HTN planning framework for agents in dynamic environments that interleaves planning and acting is presented in (Hayashi et al., 2004). In contrast to this work, the domain model is assumed to be updated by an external component and not by means of active information gathering. ENQUIRER (Kuter et al., 2004) is an HTN planning algorithm that can solve web service composition problems that require the gathering of information during the composition process. However, this approach is tailored to web service composition. A general purpose open-ended domain model and the autonomous generation of knowledge acquisition tasks are not considered.

An online planning algorithm where agents must start executing without previously generating a complete plan has been proposed by Marthi et al. (2008). Similar to the approach described in this thesis, search spaces are only partially expanded, and the prefix of partial plans are executed prior to having a complete plan. A similar approach has been proposed for probabilistic environments (Helwig and Haddawy, 1996). They extended the DRIPS (Doan and Haddawy, 1995) system to the online setup. Furthermore, Kaelbling and Lozano-Pérez (2011) propose a hierarchical planning approach that also interleaves planning and execution. In contrast to our approach, this approach does not exploit HTN methods. Instead, they construct a plan hierarchy by means of postponing the consideration of some preconditions of a planning operator.

A lot of work has been done to efficiently replan previously generated plans if the domain model is updated. For example, mars rovers have been successfully controlled by the iterative repair planning based system CASPER (Estlin et al., 2003, 2002; Chien et al., 2000). These systems are usually triggering for replanning whenever the domain model is updated. Compared to our approach, previous work on dynamic replanning focuses on efficient replanning in the light of updated domain models, but does not focus on reasoning about active knowledge acquisition.

Integrating information from external sources into the planning process is considered by (Dornhege et al., 2009; Au et al., 2004; Au and Nau, 2006; Kuter et al., 2004; Tate and Dalton, 2003), to name just a few. However, integration is not done autonomously (e.g., via autonomously integrating knowledge acquisition tasks), but mainly predefined in the domain description.

The planning system proposed in this chapter extends existing HTN planning approaches in several ways in order to deal with the fact that a lot of relevant information is not available at the beginning of the planning process. The general idea is to additionally consider ways to continue the planning process that are not known to be valid with respect to the domain model at hand, but are valid with respect to a consistent extension of the domain model. If information that is relevant for making a reasonable planning decision is not available but can be acquired, then the planner can stop the planning process and indirectly trigger the execution of necessary sensing actions so that it can continue the planning process after the

relevant information has been acquired. The underlying open-ended domain model enables the planner to determine relevant but missing information. The planner can automatically determine if and how this information can be acquired. Based on the underlying domain model it can automatically decide when it is more reasonable—or even necessary—to execute adequate knowledge acquisition tasks prior to continuing the overall planning process.

If the planner generates a complete plan, then the planning process is sound for all search spaces and complete for finite search spaces. However, for situations where the planner only generates an incomplete plan prefix, it cannot be guaranteed that there is a valid plan with this prefix. Generally speaking, we loose the general soundness of the algorithm if we drop the assumption that the planning process is monolithic. Nevertheless, the notion of a prefix-sound plan introduces a relaxed form of soundness. Furthermore, the proposed planner is integrated into a plan-based control architecture so that the overall planning and execution process is sound. How the planner is integrated into this control system is described in the following chapter.

Chapter 6

# Plan-Based Control System

*The point is that planning is not a matter of generating a program and then becoming a slave to it. It is a matter of deliberating about the future to generate a program, which need not be executed in its entirety. It might seem odd to generate an entire program and then use only an initial segment of it, but the agent is going to have to discard the plan eventually, and the generation of the entire plan legitimizes the initial segment by showing a plausible way for it to continue. A chess-playing program illustrates the point well: Such a program explores a tree of board positions, which could be thought of as containing an elaborate plan for how to respond to the opponent's possible moves. However, the program just finds the best first move and never actually attempts to remember the whole tree of board positions. It is always more cost effective to regenerate the tree after receiving the opponent's next move than to keep the old version around. The same principle applies to planning in general, except when planning is too expensive, or too little information has come in since the plan was revised.* (McDermott, 1992)

## Contents

The objective of this chapter is to describe a novel plan-based control system that enables an autonomous agent to automatically perform tasks even if it only has an incomplete, open-ended model of the world it inhabits. The proposed plan-based control system is called ACogControl. It is based on the ACogPlan planning system (see Chapter 5) and the ACogDM domain model (see Chapter 3 and 4).

The general idea of the control system is to interleave planning and acting so that missing information can be acquired by means of active information gathering. If it is reasonable to acquire additional information, then the execution of appropriate knowledge acquisition tasks is automatically integrated in the overall planning and execution process. The presented approach is intended to provide better support for situations where relevant information is not a priori available than previous approaches.

## 6.1. Architecture

This section briefly outlines robot architectures as well as introduces the architecture of the proposed control system.

### 6.1.1. Control Architectures for Robotic Agents: A Brief Overview

One of the first robot architectures is based on the *sense-plan-act* (SPA) (Murphy, 2000) paradigm. An architecture that is based on the SPA paradigm controls a robot by means of performing the following sequence of functions: sensing, planning and execution. First, the world model is updated according to the sensor data. Subsequently, a plan is generated based on the updated world model. Finally, the generated plan is executed without directly using the sensors that created the world model.

It turned out that the SPA paradigm runs into problems, since planning in real-world situations was quite time-intensive, and—more importantly—executing plans without sensing was error-prone in dynamic environments.

For the purpose of overcoming the limitations of the SPA paradigm, researchers came up with behavior-based control approaches. The general idea was to quickly generate plans that rely more directly on current sensor data. Building elaborated models of the environment should be avoided. The idea was to "[. . . ] use the world as its own model" (Brooks, 1991). Typically, behavior-based control approaches are used to achieve short-term goals in dynamic environments. Behavior-based approaches are most prominently associated with the work of Brooks (1986) and Arkin (1998). These approaches are well-suited for the implementation of reactive

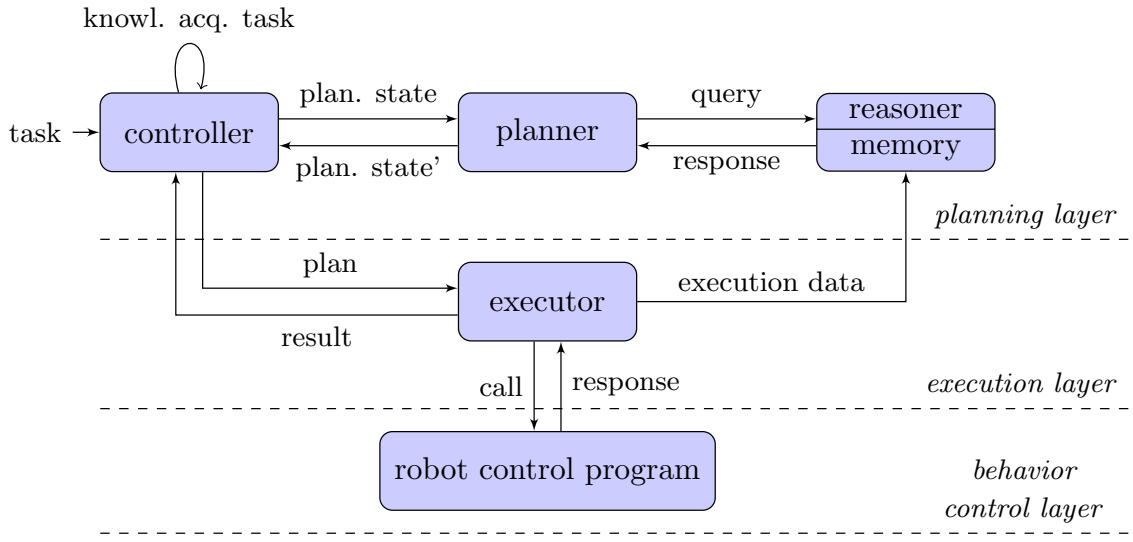



Figure 6.1.: Three tiered plan-based control architecture.

control programs that achieve short term goals, but are limited with respect to the achievement of long-term goals due to a lack of advanced reasoning and planning capabilities.

Researchers realized that robots need reactive components as well as abstract reasoning and task planning for the purpose of autonomously performing high-level tasks (e.g., "Bring me a cup of coffee"). Layered architectures were developed in order to combine the advantages of long-term AI planning and behavior-based control by means of integrating both approaches into a coherent—usually layered—architecture (Kortenkamp and Simmons, 2008).

A more comprehensive description of robot architectures can be found in Kortenkamp and Simmons (2008).

### 6.1.2. Architecture of the Proposed Control System

The proposed control system is integrated into a typical *three-tiered architecture* (Kortenkamp and Simmons, 2008) sketched in Figure 6.1. The central component of this architecture is the (plan-based) *controller*. If an agent is instructed to perform a sequence of tasks, then this sequence is sent to the controller. The controller calls the planner described in Section 5 and decides what to do in situations where the planner only returns an intermediate planning state. Furthermore, the controller invokes the *executor* in order to execute—complete or partial—plans. The executor is responsible for the execution and execution monitoring of actions. It checks the precondition of the actions prior to the execution and calls the corresponding robot control programs. In order to avoid unwanted loops (e.g., perform similar tasks more than once) it is essential to store relevant information of the execution process in the execution memory. The executor stores information about the executed actions

**Algorithm 6.1:** perform(*current_tasks*,*overall_tasks*,*domain_model*)

```
1  ps ← (domain_model, (current_tasks, <>), ∅);
2  ps' ← plan(ps);
3  if ps' is a final planning state then
4      r ← execute(ps'_p);
5      return r;
6  else
7      r ←execute(p' ⊆ ps'_p);
8      if r is a success then
9          choose ac ∈ ps'_kas with the minimum cost;
10         t_ac ← acquisition-task(ac);
11         r' ← perform(< t_ac >, overall_tasks, r_D);
12     if current_tasks only contains a (automatically created) knowledge
       acquisition task then
13         return r';
14     else
15         return perform(overall_tasks, overall_tasks, r'_D);
```

and the outcome of a sensing action in the memory system such that the domain model can properly be updated. This information includes acquired information as well as knowledge about acquisition attempts. Knowledge acquisition attempts are stored in order to avoid submitting the same query more than once to a certain knowledge source. A more detailed description of the execution memory can be found in Section 4.5. More advanced execution monitoring is out of the scope of this work and is not further addressed.

## 6.2. Algorithm

This sections describes the algorithm of the plan-based control system. The behavior of the controller is specified by Algorithm 6.1. The input of the controller is a sequence of tasks that needs to be performed, a sequence of the overall tasks, and a domain model. The sequence of the overall tasks is used as an input parameter so that the algorithm can keep track of the overall tasks by means of submitting them to recursively called instances of the algorithm. The algorithm returns an execution result. The execution result $r$ includes the resulting domain model denoted as $r_D$.

The control process begins by means of creating an initial planning state based on the given task sequence and the domain model (line 1). Subsequently, the controller submits the planning state to the planner (line 2). If the planner returns a final planning state (i.e., a planning state that contains a complete plan), then the controller directly forwards the generated plan to the executor (line 4). However, if the

planner returns an intermediate planning state (i.e., a planning state that only contains a partial plan), then the controller performs a prefix of the already generated plan (line 7), chooses the knowledge acquisition with the minimum expected cost (line 9), generates a knowledge acquisition task as defined by Definition 4.7 (line 10), and performs the knowledge acquisition task (line 11). The subsequent behavior depends on whether the task sequence at hand is a single knowledge acquisition task created by the controller itself or not. Generating a knowledge acquisition task in a concrete planning situation is a planning decision that is optimal according to the domain model at hand. In the light of additionally acquired information, however, previous planning decisions can turn out to be suboptimal. For example, let us assume that a robot has decided to first determine via perception whether `door2` is open in the situation illustrated by Figure 2.4. If the robot later finds out that it is much more difficult to acquire that information than initially expected (e.g., because a lot of objects unexpectedly obstruct the way to `door2`), then it can be more reasonable to switch to a different plan (e.g., move to the kitchen via `door1`) instead of trying to acquire the desired information at any cost. Therefore, the algorithm continues to plan the overall task after additional information has been acquired. In order to achieve this behavior, the controller (lines 12 - 13) directly returns the result of the next execution phase for generated knowledge acquisition tasks, since after the next execution phase, the controller has acquired additional information. In this way, the algorithm gives the control back to the superordinate instance of the $perform(current\_tasks, overall\_tasks, domain\_model)$ procedure (i.e., the instance that recursively called the current instance), and the controller continues to perform the overall tasks (lines 14-15). Reconsidering planning decisions after the additional information has been acquired in this way goes in line with the quotation from McDermott (1992) shown at the beginning of this chapter. The controller triggers replanning after the domain model is updated in order to not—to use the words of McDermott (1992)—become a slave of a previously generated plan.

Knowledge acquisition tasks can—like any other task—also require the performing of additional knowledge acquisition tasks which in turn require the execution of additional knowledge acquisition tasks and so on. One important feature of the proposed controller system is that it can flexibly handle this nesting of knowledge acquisition tasks. The experimental evaluation results (see Chapter 9) indicate that this is an essential feature for real-world scenarios.

Unfortunately, there is one question that cannot be considered in a completely domain-independent way: Which prefix of the already generated (partial) plan should be executed prior to the knowledge acquisition tasks (line 7)? For example, if one instructs a robot agent to deliver a cup into the kitchen, but it is unknown whether the door of the kitchen is open or closed, then it is reasonable to start grasping the cup, move to the kitchen door, sense its state and then continue the planning process. In contrast, it usually should be avoided to execute critical actions that cannot be undone until a complete plan is generated. The default strategy of the proposed controller is to execute the whole plan prefix prior to the execution of

knowledge acquisition tasks. For the service robotic scenarios and the domains from the planning community used for the experimental evaluation (see Chapter 9) this strategy worked out well. Thus, there was no need for a domain specific implementation. However, due to the fact that this is not always the best strategy, it is possible to specify domain specific control rules, or provide an alternative implementation of the controller.

## 6.3. A Complete Example

This section is intended to exemplify the behavior of the proposed control system using a simple example. Figure 6.2 illustrates the continual planning and execution process for the task of moving into the kitchen. Consider the domain model described in Section 5.3. The agent is instructed to move into the kitchen (i.e,. perform the task `move_to(kitchen)`). The proposed plan-based control system performs the task in the following eight steps illustrated by Figure 6.2:

1. The task is sent to the controller (see Algorithm 6.1). The controller instructs the planner to generate a plan for the task.

2. For the example at hand, it is assumed that the agent is close to `door2` such that moving to the kitchen via `door1`, that is known to be open, would be a heavy detour. Therefore, the planner decides to stop the planning process and returns an intermediate plan and a knowledge acquisition scheme. This knowledge acquisition scheme determines that the state of `door2` should be determined using the laser scanners as an external knowledge source.

3. The plan that is returned by the planner contains no primitive actions that can be directly executed. Therefore, the plan cannot be executed at all (cf. line 7 of Algorithm 6.1), and the controller forwards the task of determining the state of `door2` to the planner. The knowledge acquisition task is represented for reasons of simplicity without the set of derivable instances, the context, and the result (cf. Definition 4.7).

4. For this example, it is assumed that the planner can generate a complete plan for the knowledge acquisition task. It plans to approach and sense the state of `door2`.

5. The controller forwards the plan to the executor and the plan is executed. This is an example for a situation where the agent starts to execute actions before it has a complete plan for the overall task of moving into the kitchen. The agent approaches the door and determines that it is open.

6. The controller instructs the planner to continue to generate a plan for the overall task of moving into the kitchen.

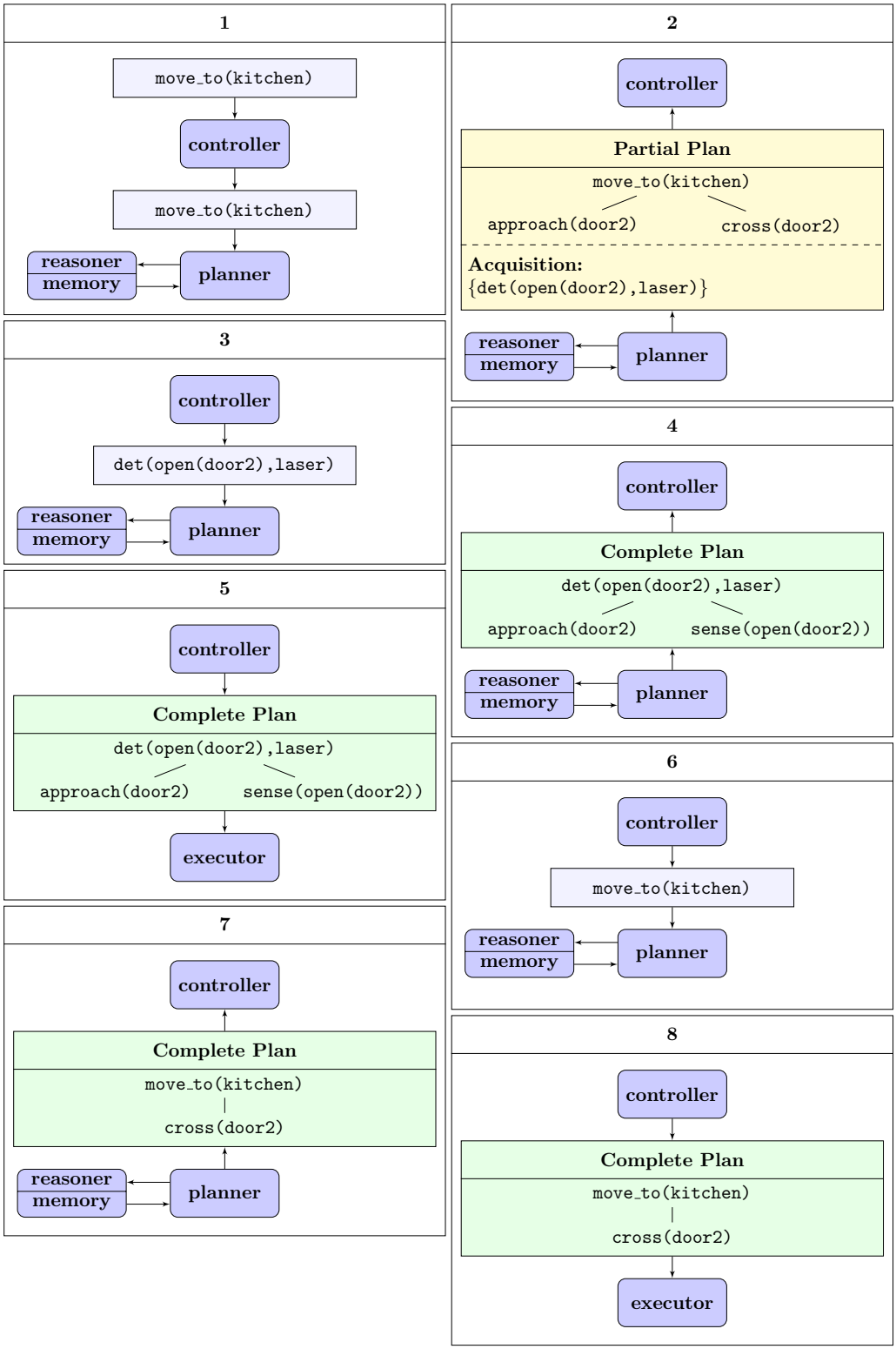



Figure 6.2.: Illustration of the continual planning and acting for the task `move_to(kitchen)`.

7. The planner can generate a complete plan. Due to the fact that the agent already approached `door2`, it only needs to cross it for the purpose of moving into the kitchen.

8. Finally, the controller forwards the plan to the executor, and the agent moves into the kitchen.

This simple example illustrates how the plan-based control system interleaves planing and acting—in particular knowledge acquisition—so that missing information can be acquired by means of active knowledge acquisition. The example is rather a simple toy example. Nevertheless, the experimental scenarios described in Chapter 9 proves that the approach can be applied to (more complex) real-world robotic domains.

## 6.4. Soundness and Completeness

Which actions are performed by the plan-based control system is decided by the underlying planning system ACogPlan. Therefore, the proposed controller is complete in the same sense and situations as the underlying planning system (see Section 5.4). This form of completeness is of course restricted to the core control system. Nevertheless, many things that are out of the scope of this work and out of the control of the proposed control system can go wrong so that a given task cannot be performed.

With respect to the soundness, one can offer an even stronger guarantee than for the underlying planning system, because if the proposed plan-based control system successfully performs a sequence of tasks, then the agent performs a sound plan due to the following reasons:

1. The underlying planning system ACogPlan is sound if a complete plan is generated (see Section 5.4).

2. If a sequence of tasks can be performed, then the last planning phase generates, according to the definition of Algorithm 6.1, a complete plan for the overall task. Thus, the last execution phase executes a valid and complete plan that performs the overall task.

Therefore, the plan-based control system is sound. Whenever the control system successfully terminates the execution of a given sequence of tasks, then it has actually performed this sequence of tasks correctly according to the domain model.

## 6.5. Multimodal Integration Processes

The integrative processing of different sensing and acting modalities is an essential approach to provide reliable information about complex, dynamic environments (Luo

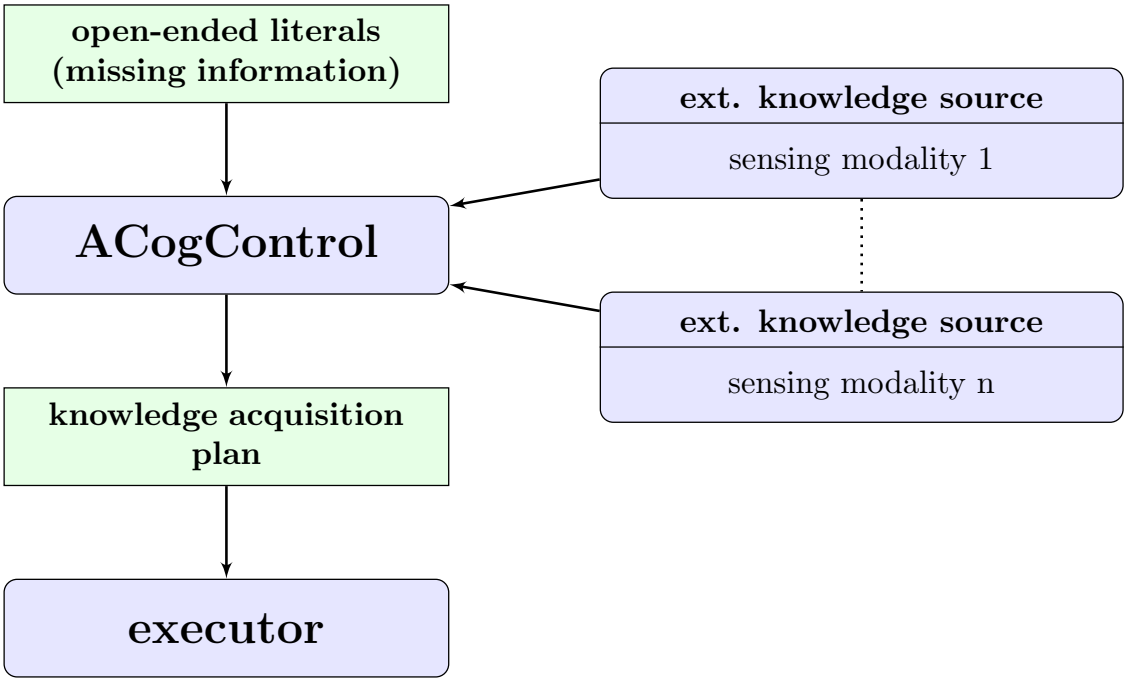



Figure 6.3.: The ACogControl system can integrate unimodal sensing actions to a multimodal knowledge acquisition plan for the purpose of acquiring necessary information. Unimodal sensing actions are viewed as external knowledge sources.

and Kay, 1995). For a robotic system, multimodal integration occurs on almost all abstraction levels.

On the behavior control level, different sensing and acting modalities are combined to multimodal robot control programs. These multimodal robot control programs can be represented by planning operators so that they can be integrated into the planning and execution process of the proposed plan-based control system. The implementation of multimodal robot control programs has the advantage that different modalities can be closely integrated. Nevertheless, the main disadvantage is that the integration is rather static and out of the control of the plan-based controller.

In addition, it is also possible to less closely but more flexibly integrate unimodal actions on the planning layer. The plan-based control system can be used to automatically integrate unimodal sensing and acting modalites such that the resulting behavior performs a task at hand. In particular, sensing modalities can be seen as external knowledge sources. If individual sensing modalities are handled as an external knowledge source, then the plan-based control system can automatically integrate them to acquire missing information. This approach is illustrated by Figure 6.3. The ACogControl system integrates unimodal sensing actions to a multimodal knowledge acquisition plan for the purpose of acquiring necessary information. In this way, the proposed control system can support the integration of sensing modalities in the following ways:

1. ACogControl can determine what information can be acquired from what sensing modality. Thus, it can link the sensing capabilities to (high-level) information that can be determined based on the sensing data. Typically, for each type of information (e.g., the relative position of an object) there is a corresponding control program that determines the desired information from the raw sensor data. The control programs are represented by a planning operator so that the plan-based control system can reason about them. How tightly different sensing modalities can be integrated usually depends on the complexity level of the actions. If the possible operations of a sensing modality are encapsulated in a larger set of less complex planning operators, then the plan-based control system is in principle able to more tightly integrate multiple sensors.

2. ACogControl can determine how expensive it is to acquire information from a certain sensing modality. Thus, in situations where the same information can be provided by several sensing modalities, the ACogControl system can choose the sensing modality which is expected to provide the best data or can more easily be used than other alternatives.

3. High-level percepts (see Section 4.9) can be used to model multimodal integration processes. The ACogcontrol system can automatically plan the execution of sensing actions and then combine the individual sensing results to a multimodal percept.

In summary, multimodal integration processes are essential for real-world robots on various abstraction levels. For the proposed plan-based control architecture, different modalities can be integrated on the behavior control level to primitive actions, and unimodal primitive actions can be automatically integrated by the plan-based controller. The latter approach particularly benefits from the reasoning capabilities and the flexibility of the ACogControl system. If the capabilities of available sensors are encapsulated in a set of sensing actions (i.e., planning operators), then the ACogControl system can actively reason about their integration. Instead of predefining a multimodal integration process, the system plans the integrative processing of sensors according to the situation at hand. This makes the approach significantly more flexible. For example, if one sensing modality is unavailable (e.g., the vision system is unavailable due to insufficient lighting conditions), then the functionality can possibly be implemented by the integrative processing of the remaining sensors. In the context of this work, unimodal sensing actions with a laser scanner and a vision system have been used for the experiments with the service robot TASER described in Chapter 9.1.

## 6.6. Discussion and Related Work

This section discusses the introduced plan-based control system and describes related work.

## 6.6.1. Related Work

This section describes related work on high-level control approaches that are intended to enable agents to perform high-level tasks (e.g., clean the table) autonomously.

### Plan-based Robot Control

Plan-based robot control refers to the usage of task planning methods in the control software of robotic agents (Hertzberg and Chatila, 2008). The integration of general purpose planning systems does not attempt to replace dedicated path and motion planning systems, but to complement them with more general reasoning and planning capabilities.

Integrating AI planning into the control system of robots has a long history. It goes back to the usage of the STRIPS (Fikes and Nilsson, 1971; Fikes et al., 1972) planning system for the plan generation procedure of the robot SHAKEY(Nilsson, 1984).

Several robotic systems including XAVIER (Simmons et al., 1997), Rhino (Beetz, 2001; Burgard et al., 1999) and Minerva (Thrun et al., 1999) successfully performed long-term demonstrations. These systems have in common that they use plans in order to improve their behavior. Beetz (2002b) developed a plan representation language for such robot systems. A more comprehensive description of plan-based control of robotic agents including the description of experimental demonstrations with Minerva and Rhino can be found in Beetz (2002a).

The ARMAR-III Asfour et al. (2006) robot was developed at the University of Karlsruhe in order to act in a household environment. The three-layered software architecture is composed of a task planning layer, a synchronization and coordination layer, and a sensor-actor layer. The planning layer decomposes abstract tasks into sets of subtasks and is responsible for the scheduling of tasks and management of resources and skills. The execution layer works in terms of the control theory and executes dedicated sensory-motor control programs. The middle layer integrates the planning and the sensor-actor layer. It invokes the subtasks from the planning layer sequentially or in parallel on the execution layer.

How a plan-based control system can benefit from the usage of additional semantic information is described by Galindo et al. (2005, 2007, 2008). This line of research integrates hierarchical spatial information and semantic knowledge into a so called *semantic map*. The information encoded in this model is exploited for the purpose of improving the task planning capabilities of a plan-based control architecture.

Müller (2008) proposed the transformational planner TRANER. The planner is used for the plan-based control of an autonomous household robot. Transformation rules transform plans of a hand-coded plan library (Müller and Beetz, 2007) to a new plan. More recent extensions of this work also consider the usage of task descriptions from the word wide web (Tenorth et al., 2010).

More recently, the the java implementation JSHOP2 (Ilghami, 2006) of the HTN

planning system SHOP2 (Nau et al., 2003) was integrated into the control architecture of mobile robots. A planning system that integrates description logic reasoning and JSHOP2 was used to control the robot "Johnny Jackanapes" (Hartanto, 2009; Hartanto and Hertzberg, 2008, 2009). In Weser et al. (2010), we described the integration of JSHOP2 into the control architecture of the service robot TASER (see Chapter 8). We proposed two implementation patterns for HTN methods in order to overcome the fact that JSHOP2 is limited by being based on the assumption that all relevant information is available at the beginning of the planning process. Compared to the plan-based control system proposed in this work, this approach is rather a workaround than a solution. The main disadvantage of the approach presented in Weser et al. (2010) is that a suitable behavior needs to be hard coded for every possible situation where possibly relevant information is not available. Unfortunately, this approach is practically impossible for realistic domains due to the sheer magnitude of situations that would have to be explicitly considered by a domain engineer.

Wolfe et al. (2010) describe the application of a hierarchical planning system to robotic manipulation. The hierarchical planning system was evaluated on a pick-and-place domain using a prototype PR2 robot constructed by Willow Garage, Inc (Wyrobek et al., 2008). Wolfe et al. (2010) demonstrated that hierarchical task planning can be extended such that it can generate high-quality plans at the level of low-level manipulation commands.

### Cognitive Robotics

The term *cognitive robotics* has been defined as:

> [...] the study of the knowledge representation and reasoning problems faced by an autonomous robot (or agent) in a dynamic and incompletely known world. Central to this effort is to develop an understanding of the relationship between the knowledge, the perception, and the action of such an robot. (Levesque and Reiter, 1998)

According to this definition, a major part of this work can be located in the area of cognitive robotics. Cognitive robotics approaches typically use logic as the medium of knowledge representation and theorem proving as the universal reasoning process (Shanahan, 2000; Shanahan and Witkowski, 2001).

The Golog family of action languages have received much attention in the cognitive robotics community (Levesque et al., 1997; Tam et al., 1997; Lakemeyer, 1998). They are based on the situation calculus (Reiter, 2001). The basic Golog version has been extended in various ways. ConGolog (Giacomo et al., 2000) additionally considers concurrency, interrupts, and exogenous actions. The problem of performing tasks in open-ended domains is most extensively considered by the *IndiGolog* language (Giacomo and Levesque, 1999; Sardiña et al., 2004), since programs are executed in an online manner, and thus the language to some degree is applicable to situations

where the agent has only incomplete information about the state of the world. Regrettably, IndiGolog only supports binary sensing actions.

Another interesting high-level control framework is *READYLOG* (Ferrein and Lakemeyer, 2008). READYLOG is a Golog dialect which was developed to support high-level robotic control in highly dynamic domains (e.g., robotic soccer). In contrast to READYLOG, which mainly supports passive sensing, ACogControl focuses on the active sensing of unknown information.

Another powerful agent programming language is *FLUX* (Thielscher, 2005a,b) which is based on the *Fluent Calculus* (Thielscher, 1998). FLUX (Thielscher, 2000) is a powerful formalism but uses a restricted form of conditional planning. As already pointed out, conditional planning is not seen as an adequate approach for the scenarios we are interested in. A more detailed overview of logic-based control for robots can be found in Levesque and Lakemeyer (2008).

### 6.6.2. Discussion

This chapter has presented a new plan-based control system. The control system is based on the new open-ended domain model ACogDM (see Chapter 3 and 4) and the new HTN planning system ACogPlan (see Chapter 5). It constitutes a continual planning and execution system that automatically interleaves planning and execution. If relevant information is not available, then the controller automatically integrates the execution of corresponding knowledge acquisition tasks.

The proposed plan-based control system is more flexible and can perform tasks in a larger set of situations than existing approaches, that are based on classical planning (e.g., Brenner and Nebel (2009); Goebelbecker et al. (2011); Talamadupula et al. (2010a)), since it can exploit the advanced reasoning capabilities of the underlying planning system. For example, it can perform tasks in situations where no factual knowledge is a priori available (see Chapter 9), or situations where the execution of a knowledge acquisition task requires the execution of an additional knowledge acquisition task, and so on.

Like previous HTN planning systems, the proposed plan-based control system is *domain-configurable* (Nau, 2007). Thus, the core planning, reasoning and controlling engines are domain independent, but can exploit domain specific information. The proposed control architecture can be viewed as a general framework that features the automatic as well as manual integration of external components.

The control system can release domain engineers from the burden of explicitly dealing with the fact that often a lot of information is not initially available, because it automatically integrates the execution of appropriate knowledge acquisition tasks. The following chapter provides several examples from the service robotic domain that illustrate how a domain model can be specified.

Chapter 7

# Representing Knowledge in Open-Ended Robotic Domains

*As far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality.* (Albert Einstein, 1879-1955)

## Contents



This chapter describes how one can represent a domain model for a typical service robotic domain, whereby the goal of this chapter is twofold:

1. It is intended to demonstrate how some of the described features of the ACog-Control framework can be used to represent knowledge for a typical service robotic domain. In particular, the goal is to point out that the proposed architecture can release a domain engineer from the burden of explicitly dealing with the fact that often not all relevant information is available at the beginning of the planning and execution process.

2. The proposed control system is evaluated, beside other domains, on a physical service robotic system (see Chapter 9). The second main objective of this chapter is to describe a representative subset of the domain model used for these experiments.

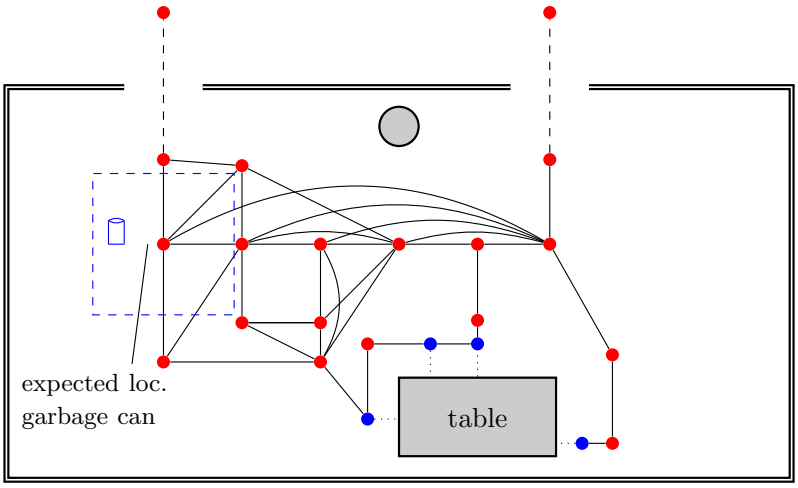



Figure 7.1.: Illustration of the experimental environment including the navigation graph.

## 7.1. Continual Path Planning

This section describes how the proposed plan-based control system can be used for continual path planning.

### 7.1.1. General Approach

The domain model of the service robot domain contains a *navigation graph* (see Figure 7.1) that determines which waypoints the robot can directly reach from a given waypoint without contacting a static part of the environment (e.g., a wall or a table). It is based on the concept of a *roadmap* (Latombe, 2003; LaValle, 2006) and is used for (local) path planning inside a room. However, in the open-ended environments this work is interested in, possibly unknown obstacles between two connected waypoints can obstruct the direct reachability. Path planning in open-ended domains provides an interesting test case for the proposed plan-based control system. Continual planning is a suitable approach for path planning in these situations where no complete model is available, because it creates a path incrementally considering relevant sensing data. In other words, instead of generating a complete—possibly incorrect—navigation path in advance, the next traversed waypoint is chosen by the continual planner only if it has been sensed that nothing obstructs a passage. One of the key benefit of the proposed continual, plan-based control system is the fact that a continual interleaving of planning and acting (e.g., sensing) is achieved automatically. This significantly eases the knowledge engineering process, since a domain engineer does not have to explicitly describe when to acquire what information (e.g., determine whether it exists a free path between to waypoints of the

```
method(navigate(_,_,To),
  % precondition
  at_wp(robot,To),
  % subtasks
  [],
  % expected cost
  0).
method(navigate(_,From,To),
  % precondition
  (at_wp(robot,From), nav_edge(From,To),
    free_path(From,To)),
  % subtasks
  [traverse(From,To)],
  % expected cost
  eval(Cost,dist(From,To,Cost))).
method(navigate(Visited,From,To),
  % precondition
  (at_wp(robot,From), nav_edge(From,Mid), Mid \= To,
    Mid notin Visited, free_path(From,Mid)),
  % subtasks
  [traverse(From,Mid), navigate([Mid|Visited],Mid,To)],
  % expected cost
  eval(Cost,nav_cost(From,Mid,To,Cost))).
```

Figure 7.2.: HTN methods that describe how the robot can navigate to a desired waypoint.

navigation graph) prior to continuing the overall planning process. Therefore, HTN methods can be defined as if necessary information is always available, since the proposed control system deals with the described challenges of the open-endedness automatically.

Let us consider the path-planning task as an example and see how one can define HTN methods for this task. There are many different ways to model path planning with the proposed planning-based control framework. Figure 7.2 shows a simple representation used the for the experiments presented in Chapter 9. Three HTN methods for the task `navigate(Visited,From,To)` describe the three distinguished situations:

1. The robot is already at the goal position. No further planning is necessary and the cost is zero.

2. The robot is on a waypoint that is, according to the navigation graph, directly connected with the goal waypoint (`nav_edge(From,To)`), and the path to the goal position is free (`free_path(From,To)`). The robot can directly traverse to

the goal position. The cost is defined as the distance from the current position to the goal position.

3. There is a waypoint `Mid` that is, according to the navigation graph, directly connected with the current waypoint (`nav_edge(From,Mid)`), not the goal waypoint (`Mid \= To`), and the path from the current waypoint to `Mid` is free. For the purpose of avoiding it to end up in an infinite loop, the method keeps track of visited waypoints and ensures that a single waypoint is at most considered once. The planner chooses an intermediate waypoint that fulfills the aforementioned constraints and continues the navigation from this point. The cost is defined using the derived predicate `nav_cost(From,Mid,To,Cost)` described below.

In order to find a good path fast, the planner needs a heuristic so that it can reasonably choose the next waypoint. Technically speaking, it needs an approach to derive an instance of `nav_cost(From,Mid,To,Cost)`. A comparably simple approach is to use the sum of the euclidean distance from the current waypoint to the intermediate and the intermediate to the goal waypoint as the heuristic function. This heuristic is represented by the first domain specific axiom shown in Figure 7.3. A computational harder but usually better heuristic is represented by the second axiom. The general idea of the second heuristic is to use a fast (i.e., non-optimal) planning procedure for the purpose of estimating the cost of navigating to a desired goal position via a certain intermediate waypoint. Technically, this is achieved by using the **`call`** meta-predicate in order to call ACogPlan in the *atomic* mode. With the **`call`** meta-predicate it is possible to call arbitrary external programs (see Section 3.7.2). In the *atomic* mode the planner generates a complete plan (i.e., there is only one planning phase) or returns a failure. Hence, the planning phase constitutes an atomic process. The planner runs faster in the atomic mode than in the "normal" (i.e., continual) mode, since it does not have to consider the integration of knowledge acquisition tasks. The **`dm`** meta-predicate provides access to the domain model so that it can be passed to the planner (see Section 3.7.6).

Figure 7.4 shows the methods for the `route(Visited,From,To,Route)` task used by the second axiom shown in Figure 7.3. Like for the `navigate` task, three methods distinguish between the case where the goal position is already reached, the goal position is directly connected to the current position, or an intermediate waypoint is selected and planning continues from this intermediate waypoint.

Compared to the `navigate` task, it is only necessary that a free path between two waypoints possibly exists. Possibly true statements are first-class citizens in the proposed planning framework. The possibly existing free path can be expressed using the **`possibly`** meta-predicate (see Section 3.7.7) as shown in Figure 7.4. In this way, the planner will only consider adjacent waypoints if there possibly is a free path segment (i.e., it is not known that such a free path cannot exist).

```
axiom(nav_cost(From,Mid,To,Cost),
  (dist(From,Mid,Dist1),dist(Mid,To,Dist2),
  Cost is Dist1 + Dist2)).
axiom(nav_cost2(From,Mid,To,Cost),
  (dist(From,Mid,Dist1),dm DomainModel,
  call((
    ground(Mid),
    ground(To),
    planner::plan_atom(DomainModel,
      [route([Mid],Mid,To,_)],P),
    P::resulting_cost(Dist2),
    Cost is Dist1+Dist2)))).
```

Figure 7.3.: Axioms that define two possible heuristics for the path planning.

### 7.1.2. Compared to Existing Path Planning Approaches

The presented representation of path planning was successfully used for the experiments presented in Chapter 9. However, compared to many existing approaches, the presented representation of path planning is rather simple. It exists a huge amount of literature for specific path planning and search algorithms that have been successfully applied to solve real-world path planning problems. Most notably this includes incremental heuristic search like *focused D** (Stentz, 1995) or *D* lite* (Koenig and Likhachev, 2002), which are often used for robotic path planning in unknown terrain. They are able to replan—which is necessary if unforeseen objects make a previously generated plan impossible—quickly, because they modify previous search results only locally. Furthermore, previous work includes approaches that can deal with non-deterministic (e.g., feedback motion planning (LaValle, 2006)) action primitives.

One of the main disadvantage of the aforementioned, dedicated path planning approaches is the fact that they only have limited reasoning capabilities if their are not more tightly integrated into a task planning environment. A task planner usually calls a path planner as a subroutine in a top down manner (Koenig, 2010). However, it can also be beneficial—or even necessary—to exploit the advanced reasoning capabilities of a task planner in a bottom-up way.

For example, in addition to generate an alternative path in a situation where a previously generated plan turns out to be incorrect (e.g., because an unexpected object obstructs a passage), an agent can also use one of the following strategies:

1. It can actively free the path (e.g., move objects aside that obstruct a passage).

2. It can switch to a different plan on a superordinated abstraction level (e.g., switch to a different goal position).

```
method(route(_,To,To,[]),
  % precondition
  true,
  % subtasks
  [],
  % expected cost
  0).
method(route(Visited,From,To,[To]),
  % precondition
  (nav_edge(From,To), To notin Visited,
    possibly free_path(From,To)),
  % subtasks
  [do_route(From,To)],
  % expected cost
  eval(Cost,dist(From,To,Cost))).
method(route(Visited,From,To,[Mid|RemRoute]),
  % precondition
  (nav_edge(From,Mid), Mid notin Visited,
  possibly free_path(From,Mid)),
  % subtasks
  [do_route(From,Mid),route([Mid|Visited],Mid,To,RemRoute)],
  % expected cost
  eval(Cost,(dist(Mid,To,Dist1),
    Cost is Dist1 + 10000))).
```

Figure 7.4.: Methods used to quickly find a path to a goal waypoint and thereby providing a heuristic for the expected cost.

### 7.1.3. Advantages of the Proposed Approach

This sections exemplifies the two aforementioned strategies for a situation where a previously generated plan turns out be incorrect in the light of new information.

#### Automatically Considering Additional Planning Alternatives

Using the proposed framework for path planning has the advantage that the path planning process automatically benefits from the available reasoning and planning capabilities. For example, in order to consider the fact that the robot is able to free an impassable path (e.g., via moving an object that obstructs a passage aside), we can add two methods for the `navigate(Visited,From,To)` task (as shown in Figure 7.5) to the domain specification. In this example, 5000 is defined as the cost for the `free_path(W1,W2)` task. If these two methods are part of the domain model, then the plan-based controller automatically considers the possibility to free the path. Whenever reasonable (e.g., because no additional path exists or all alternative paths would be a heavy detour), the robot frees the path instead of trying to generate an

```
method(navigate(_,From,To),
  % precondition
  (at_wp(robot,From), nav_edge(From,To),
      neg free_path(From,To), can_free_path(From,To)),
  % subtasks
  [free_path(From,To),traverse(From,To)],
  % expected cost
  eval(C,(dist(From,To,C1),C is C1 + 5000))).
method(navigate(Visited,From,To),
  % precondition
  (at_wp(robot,From), nav_edge(From,Mid), Mid \= To,
      Mid notin Visited, neg free_path(From,Mid),
      can_free_path(From,Mid)),
  % subtasks
  [free_path(From,Mid),traverse(From,Mid),
    navigate([Mid|Visited],Mid,To)],
  % expected cost
  eval(C,(nav_cost(From,Mid,To,C1),C is C1 + 5000))).
```

Figure 7.5.: Two additional methods for the `navigate(Visited,From,To)` task that consider the fact that the robot can sometimes actively free the path.

alternative path. If the robot does not known whether it can free the path in a given situation, then it also takes the cost to figure it out into account. In situations where it is more reasonable to try to move the object aside, ACogControl automatically stops the initial planning process, creates an appropriate knowledge acquisition task, and determines whether it can actually free the path (e.g., via figuring out whether it can pick up the object). If it is possible to free the path, then the robot frees the path and continues to perform the overall task. This example demonstrates how path planning benefits from being integrated into the overall control framework. Dedicated path planning approaches usually are less flexible.

### Switch to a Different Plan on a Higher Abstraction Layer

Another important issue that is considered by the proposed path planning approach is the fact that if a new path must be generated (e.g., because unforeseen objects make the initially generated plan impossible), then this usually does not only have an impact on the path planning layer but also on the task planning layer. If a previously generated path to a desired goal position turns out to be impossible in the light of new information, then the cost—or even the general feasibility—of superordinated tasks changes. For example, consider a situation where we instruct a robot to pick up an object from a table. There many possible poses—usually an uncountably infinite number—from which the robot can pick up the object. Figure 7.6 shows the definition of the HTN methods for the `pick_up(Object)` task used for

```
method(pick_up(Object),
  % precondition
  in_hand(Object),
  % subtasks
  [],
  % expected cost
  0).
method(pick_up(Object),
  % precondition
  (free(hand), small_object(Object), table(Table1),
    on(Object,Table1), at_wp(robot,From), From \= unknown,
    approach_pose(Table1,pose(Waypoint,Angle)),
    pos(Object,pose(Waypoint,Angle),X,Y),
    reachable_with_arm(X,Y),
    nav_cost(From,From,To,RouteCost)),
  % subtasks
  [ navigate(pose(Waypoint,Angle)),
    grab(Object,pose(Waypoint,Angle))],
  % expected cost
  RouteCost).
```

Figure 7.6.: HTN method definitions for the task `pick_up(Object)`.

the experimental evaluation described in Chapter 9. The cost of approaching a pose from which the robot can reach the object according to its kinematic constraints is defined as the expected length of the navigation path to that pose. However, in situations where the robot senses an a priori unknown obstacle, a previously chosen pose may not be the one with the shortest navigation path anymore. In such a situation, it can be reasonable to approach a table from a completely different direction. In other words, it can be more reasonable to switch to different plan above the path planning layer. Hence, instead of finding a different path to the previously selected pose, the proposed planner automatically chooses a different pose if this results in a better plan. This is particularly relevant for a situation where the previously chosen pose cannot be reached at all. Depending on the domain model and the overall tasks of the robot, it can be reasonable to switch to a different plan that even more significantly differs from the previously chosen one. In principle, the planner can go up the abstraction hierarchy constituted by the HTN methods up to the level of the overall tasks in order to change previously made planning decisions.

## 7.2. Pick Up an Object From a Table

Let us now look more closely at the HTN methods for the `pick_up(Object)` task defined in Figure 7.6. According to the definition of the HTN methods, the robot

needs to know a reachable pose and the exact relative position of the object regarding to this pose. The relative position of the object must enable the robot to reach the object according to the kinematic constraints of the manipulator. All these conditions are represented by the precondition of the second method shown in Figure 7.6. The specifications of the HTN methods for the `pick_up(Object)` task constitutes another example for the fact that HTN methods can be defined as if all necessary information is available at the beginning of the planning process. A domain engineer does not have to consider which of this information is available, or needs to be acquired for the purpose of performing the task according to relevant HTN methods, since the proposed control framework integrates necessary knowledge acquisition tasks automatically. For example, the control architecture automatically integrates knowledge acquisition tasks that determine the exact position of the object and a reachable pose from which the object can be picked up according to the kinematic constraints of the manipulator if this information is not initially available, but constitutes a possible extension of the domain model at hand.

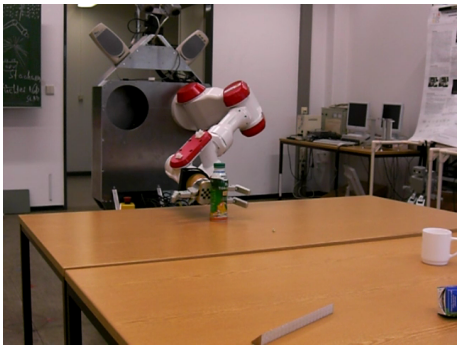
For example, the plan-based control architecture enables the service robotic system TASER to pick up an object from a table without knowing the location in a variety of situations as illustrated by Figure 7.7. TASER flexibly sensed the position of the object on the table and found a pose from which it could pick up the object. The mobile service robot platform TASER is described in more detail in Chapter 8.

## 7.3. Concluding Remarks

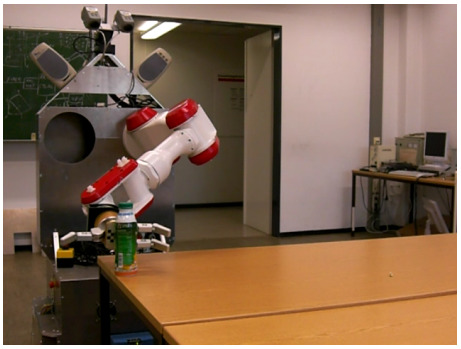
This chapter has exemplified how a domain model can be specified for the two typical service robotics tasks of moving to a goal pose and picking up an object from a table. For example, it has been described how continual path planning can be achieved. The resulting path planning approach generates a plan as far ahead as the robot can perceive the environment. The main advantage of the proposed approach is the fact that the continual planning approach automatically benefits from the proposed features of the plan-based control system. Section 7.1.3 illustrated that the continual path planning approach profits from being integrated into the overall control system by means of being able to consider additional ways to deal with a situation where a previously generated plan turns out to suboptimal after the domain model is updated. In contrast to dedicated path planning approaches, the proposed approach additionally considers it to actively free the path or switch to a different plan on a higher abstraction level.

Furthermore, this chapter has pointed out that HTN methods can be specified as if all information is available, since the plan-based controller detects relevant but missing information automatically and integrates the execution of appropriate knowledge acquisition tasks. In this way, domain engineers can be released from the burden of dealing with possibly missing information explicitly.

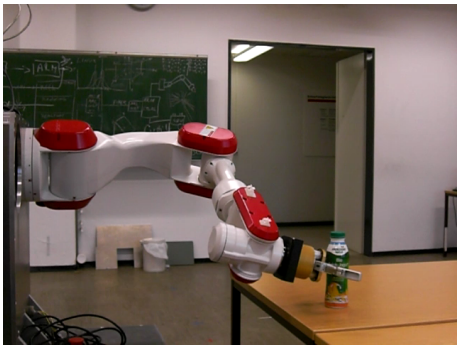
In addition to exemplifying some of the features of the ACogControl system, the domain model specifications that have been presented in this chapter are intended



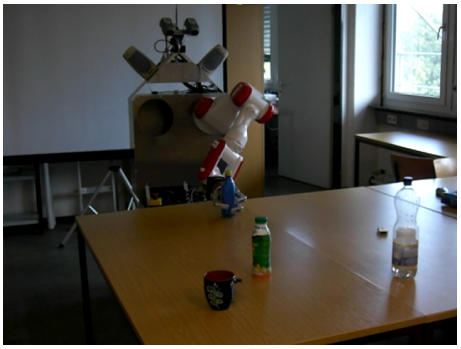
(a) TASER picks up an object located at the center of the table.



(b) TASER picks up an object located at the front right of the table.



(c) TASER picks up an object located at the bottom right of the table.



(d) TASER picks up an object located at the bottom left of the table.

Figure 7.7.: The proposed plan-based control system can perform the task of picking up an object from a table in a variety of situations without a priori knowing the location of the object. The only knowledge available about picking up an object from a table is shown in Figure 7.6.

to constitute a representative subset of the domain model used for the experimental evaluation with the service robot TASER (see Chapter 9). The complete specification of the domain model can be found in Appendix B. For clarity reasons, the domain specifications presented in this chapter differ slightly from the specifications shown in Appendix B. However, these differences have no significant impact on the semantics of the domain model.

Chapter 8

# Experimental Platform

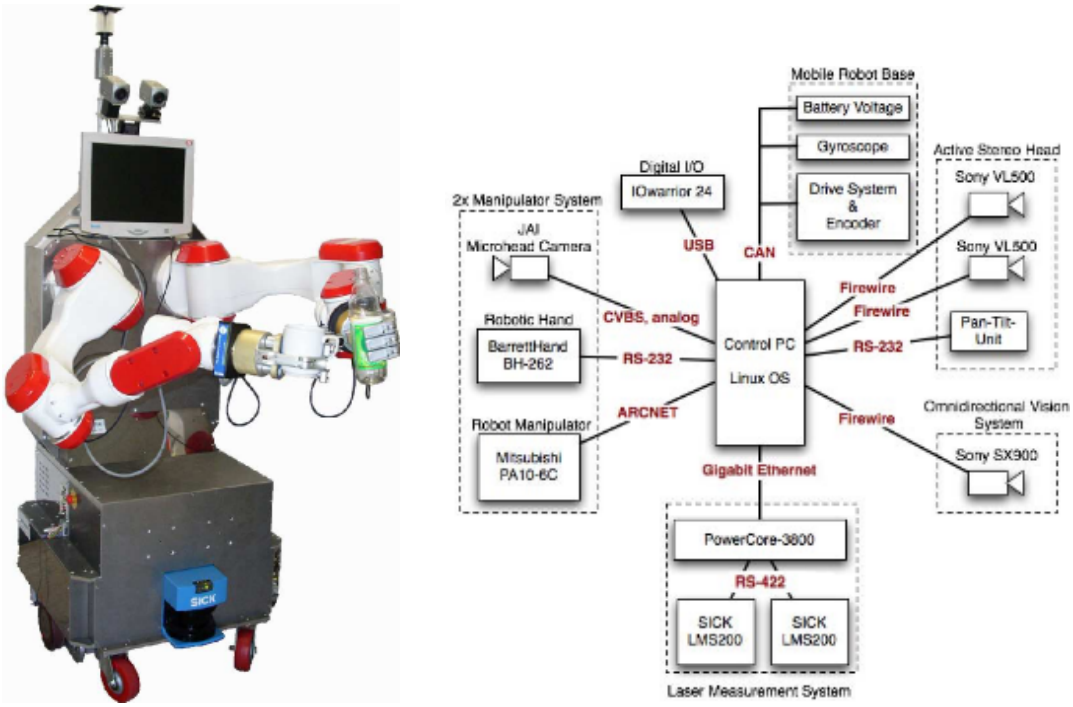
*[...] there is a tendency in the field to praise hardware and condemn software. Hardware is speedy and reactive; software spends its time swapping and garbage collecting. This preference is a temporary aberration, I believe, based on a misunderstanding about programming. As in the rest of computer science, robotics can't escape the advantages of first expressing behaviors as textual objects and later worrying about mapping them to hardware.* (McDermott, 1992)

## Contents



This chapter describes the multimodal service robot platform *TASER (TAms SErvice Robot)* shown in Figure 8.1a. The ACogControl System is implemented on TASER. TASER serves as the experimental platform for the real-world experiments described in Section 9.1. The main part of this work is independent of the presented robot platform. It can be applied to other robots or software agents as well. However, the service robotic platform TASER provides an excellent testbed for the proposed plan-based control architecture. It makes it possible to evaluate the performance characteristics of the ACogControl system in a real-world service robotic context.

Additional information of the experimental platform can also be found in Baier-Löwenstein (2008); Weser (2010); Jockel (2009).




(a) TASER in the aspired two-arm configuration. Source: (Baier-Löwenstein, 2008).

(b) Input/Output infrastructure of TASER. Source: (Jockel, 2009).

Figure 8.1.: The service robot TASER.

## 8.1. Hardware

This section describes the hardware components of the service robot TASER that were used in the context of this work.

### 8.1.1. Basic Platform

The robot platform is based on the mobile platform Neobotix MP-L655. The platform contains eight lead-gel-batteries, the control PC, and several hardware-specific controllers. The batteries supply a main power of 48 volts and a total power of 3.84 kWh. The robot can work up to eight hours using the batteries. The mobility is provided by three passive and two active wheels. An upper body is mounted on top of the mobile platform. The upper body is constructed to host two robot arms. In the current configuration, however, only one arm is mounted. The measurements of the basic platform are shown in Figure 8.2. The robot system is controlled by a single Pentium IV 2.4 GHz industrial computer with 1 GB memory. The hardware components of the robot are connect as shown in Figure 8.1b.

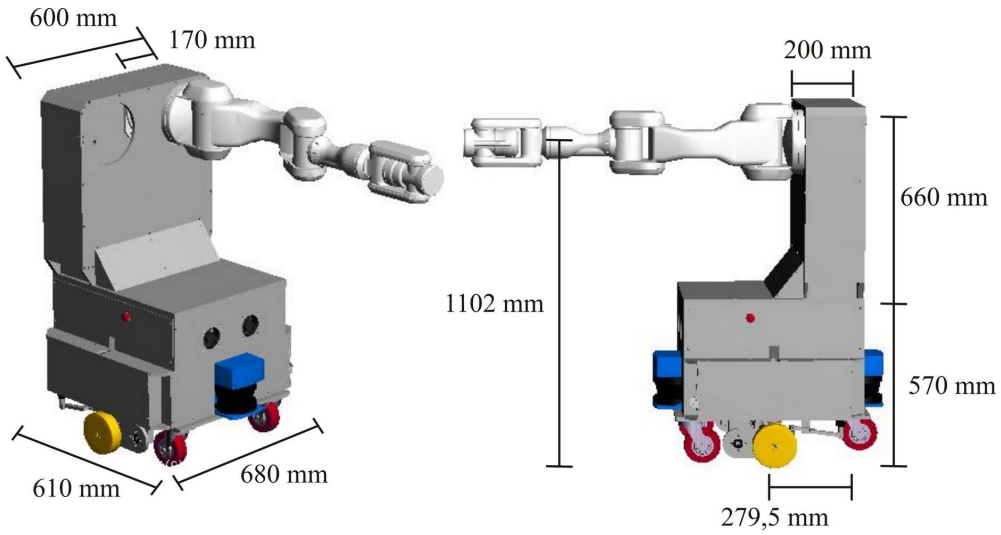



Figure 8.2.: Dimensions of the robot platform with one arm mounted. Source: (Baier-Löwenstein, 2008).

### 8.1.2. Manipulator

TASER is equipped with a manipulator consisting of a six degrees of freedom (DoF) Mitsubishi PA10-6C arm manufactured by Mitsubishi Heavy Industries, Ltd.[1] and a three-finger hand by Barrett Technologies Inc[2].

The Mitsubishi PA10-6C arm is commercially available. The dimensions of the arm are shown in Figure 8.3. It has a total length of 1317 mm. Furthermore, the arm has a weight of 39 kg and a maximum payload of 10 kg. It has six DoF and a working range that is comparable to a human arm with seven DoF. A picture of the arm is shown in Figure 8.4a, and the alignment of the axes is shown in Figure 8.4b. The robot arm is controlled by means of using the *Robot Control C Library* (RCCL) that is written by Lloyd and Hayward (1992) and adapted to the PA10 series by Scherer (2004).

The BarredHand BH8-262 is commercially available by Barret Technologies Inc. It is connected to the control PC via RS-232 and mounted as an end effector on the Mitsubishi PA10-6C robot arm. The dimensions are shown in Figure 8.5. The two joints of the fingers are controlled by one motor by means of a special gear box called *TorqueSwitch*$^{TM}$ (Townsend, 2000). It allows a constrained control of two joints with one motor. If the first link of a finger (joint $J12$, $J22$, and $J32$ in Figure 8.5) realizes resistance, then the motor stops it and drives the second link (joint $J13$, $J23$ and $J33$ in Figure 8.5) until it meets resistance too.

The payload of the whole manipulation unit (i.e., arm and hand) is approximately 4 kg.

[1] `http://www.mitsubishi-heavy.de/`, last accessed February 20, 2012.
[2] `http://www.barrett.com/`, last accessed February 20, 2012.

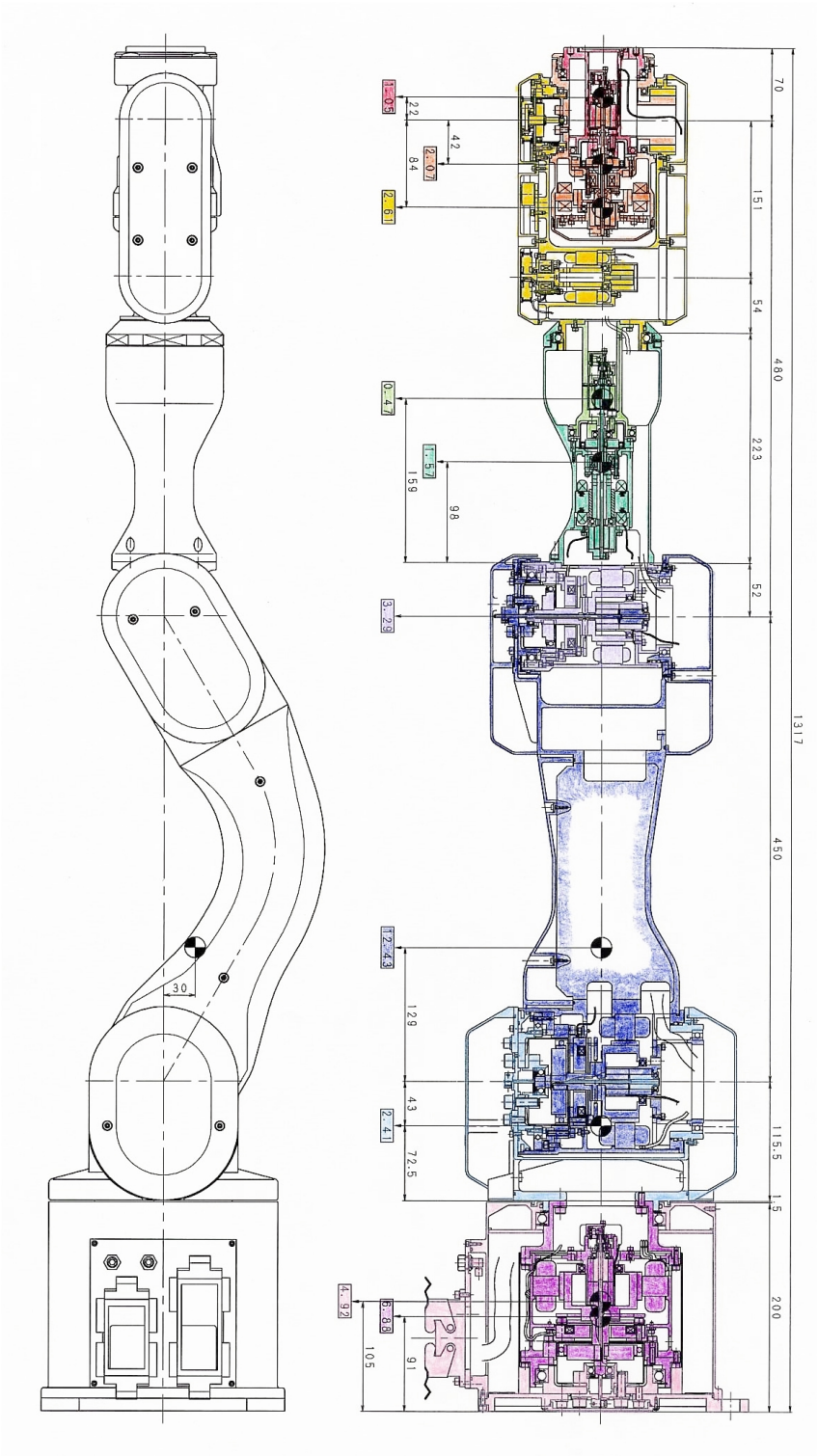
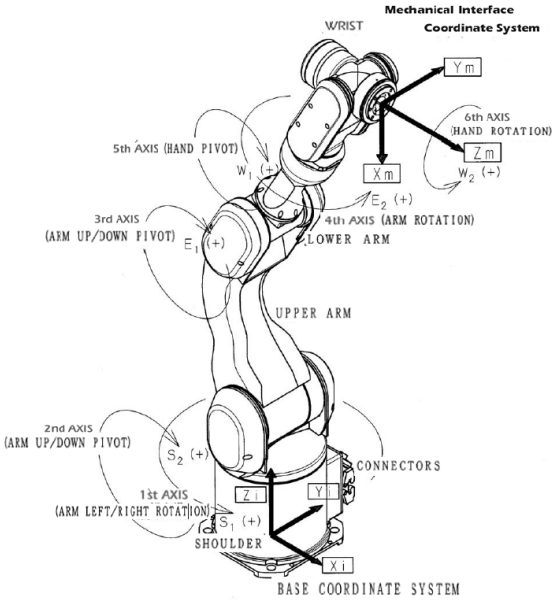



Figure 8.3.: Dimensions of the PA10-6C robot arm by Mitsubishi Heavy Industries, Ltd. Source: (Jockel, 2009).

(a) Picture of the PA10-6c. Source: (Weser, 2010).




(b) Alignment of PA10-6C axes. Source: (Mitsubishi heavy industries, 2002).

Figure 8.4.: PA10-6C robot arm by Mitsubishi Heavy Industries Ltd.

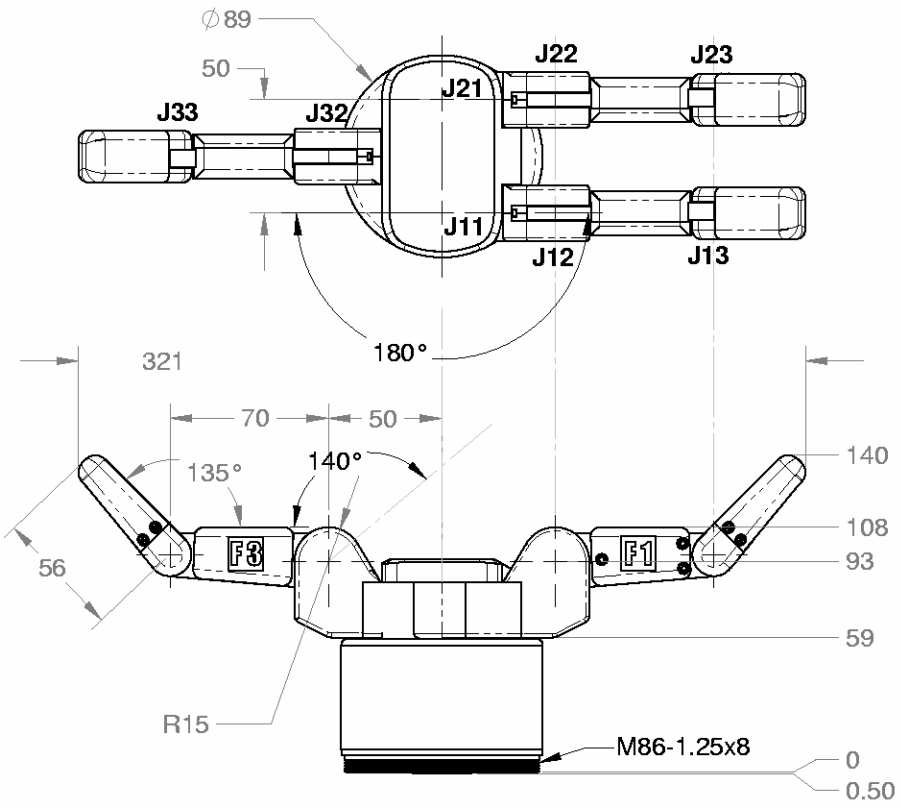



Figure 8.5.: Metric dimensions of the BarrettHand BH8-262 in *mm* and the joint radii. Source: `http://www.barrett.com`.

### 8.1.3. Sensors

For the purpose of perceiving its environment the robot is equipped with several sensors. Two SICK[3] Laser Measurement Sensors (LMS) 200 are mounted at the front and the back of the mobile platform. Furthermore, the robot has two Sony DFW-VL 500 cameras mounted on a pan-tilt-unit PTU-46-17.5 (by Directed Perception[4]) on top of the robot. The cameras are IEEE DC1394 cameras and have a maximum resolution of 640x480 px at a frame rate of 30 frames per second in YUV411. The vision system is shown in Figure 8.6. The omnidirectional camera shown in Figure 8.6 has not been used for this thesis.

## 8.2. Software

The software architecture of the service robot TASER is based on the *Roblet* (Westhoff et al., 2006) client-server architecture. The Roblet framework supports the flexible development of distributed systems. Although its application is not limited to

[3] `http://www.sick.com`, last accessed February 20, 2012

[4] `http://www.dperception.com`, last accessed February 20, 2012.

Figure 8.6.: Vision system of TASER consisting of Sony DFW-VL 500 mounted on the pan-tilt-unit PTU-46-17.5 and an omnidirectional camera. Source: (Jockel, 2009).

robotics, the provided features are well-suited for distributed robotic systems. The Roblet framework makes it possible to develop, compile, and execute an application on one workstation. During the execution it sends part of itself to corresponding servers so that the execution is reasonably distributed.

For the service robot TASER, the Roblet technology is used to provide a coherent software infrastructure. The whole Roblet framework is implemented in Java. Native hardware drivers are integrated using the Java native interface (JNI).

For this work, the Roblet framework is used to implement the robot control programs of the primitive robot actions. A description of the set of primitive actions used for this work can be found in Section 9.1.2. An overview of the control architecture is shown in Figure 8.7.

## 8.3. Concluding Remarks

In this chapter, the service robot platform TASER has been briefly described. TASER consists of a set of complex hardware and software systems. The reader has been referred to additional literature that provides more detailed information about the individual components.

The described service robot TASER was used for the real-world experiments described in Section 9.1.

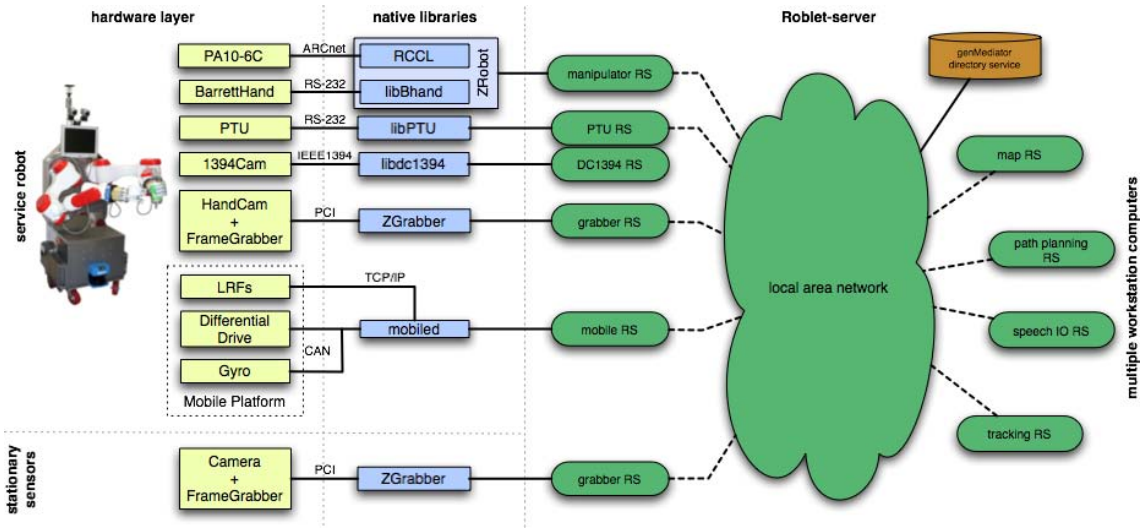



Figure 8.7.: The underlying software architecture of the robot TASER. The Hardware components are shown in yellow boxes, corresponding native (C/C++) libraries have blue boxes and the Roblet servers are green. Source: (Baier-Löwenstein, 2008).

Chapter 9

# Experimental Evaluation

*A theory is something nobody believes, except the person who made it. An experiment is something everybody believes, except the person who made it.* (Albert Einstein, 1879-1955)

## Contents



This chapter describes the experimental evaluation of the overall plan-based control system ACogControl. The control system is implemented and evaluated on the mobile service robot TASER (see Chapter 8) as well as using a set of simulated domains.

For all domain models that were used for the experimental evaluation, HTN methods and planning operators are defined as described in Chapter 7. Hence, they are defined as if all information were available at the beginning of the planning process. The domain model does not contain domain specific knowledge that directly describes when or which information should be acquired. The general purpose planning and control framework is supposed to automatically determine when it is more

reasonable to acquire additional information prior to continuing the planning process, and what kind of information and how this information should be acquired in order to reasonably perform a task.

## 9.1. Evaluation on Physical Service Robot

This chapter describes the experiments that were conducted with the service robot TASER.

The experiments were conducted in a laboratory of the University of Hamburg. The laboratory is used by various students that work on different hard- and software projects. There are no special regulations with respect to the tidiness. Therefore, the environment actually is an example for an unstructured, natural, human environment.

A simple test-case using a preliminary version of the proposed control system was already described in (Off and Zhang, 2011a).

### 9.1.1. Domain Model

A domain model of the robot TASER and the experimental environment was created that models the capabilities of the robot as well as the state of the environment. Some parts of the domain model were described in Chapter 7. The complete domain model specification can be found in Appendix B.

Table 9.1 summarizes the quantitative characteristics of the domain model instance used for the experiments with TASER. The state model contains 112 facts, whereby 95 of these facts have the interpretation model *owa* and 17 have the interpretation model *cwa*. Moreover, the domain model instance contains 22 domain specific rules and 25 action primitives. Six of the 25 action primitives are sensing actions. Furthermore, the domain model contains 68 HTN methods for 39 nonprimitive tasks. Thus, it contains on average $\frac{68}{39} = 1.74$ HTN methods per nonprimitive task and $\frac{68}{25} = 2.72$ HTN methods per action.

### 9.1.2. Library of Primitive Robot Actions

Table 9.2 describes all actions that are used for the real-world experiments with the service robot TASER. A corresponding robot control program is implemented for every primitive action. Some actions have been implemented in the context of this work, whereas other have been implemented by Weser (2010).

The controller of the mobile platform provides a collision avoidance function that is enabled by default. If the collision avoidance feature is enabled, then the mobile platform is automatically halted in front of obstacles so that no collision occurs. All actions of the mobile platform including

- `approach_position(X,Y)`,

| | |
|---|---|
| **Facts** | 112 |
| **OWAFacts** | 95 |
| **Domain Specific Rules** | 22 |
| **Action Primitives** | 25 |
| **Sensing Actions** | 6 |
| **HTN methods** | 68 |
| **Nonprimitve tasks** | 39 |
| **Knowledge Acquisition Tasks** | 7 |

Table 9.1.: Summary of the domain specification used for the experiments with service robot TASER.

- `approach_pose_planned(X,Y,Degree)`,
- `approach_pose_direct(X,Y,Degree)`,
- `move_forward_no_ca(Distance)`,
- `release_brakes`,
- `apply_brakes`,
- `move_towards(Distance,Degree)`,
- `rotate_rel(Degree)`,
- and `rotate_towards(X,Y)`

are executed asynchronously. Thus, for these actions, the executor does not wait until the execution of an action is finished. Another action can be started such that both actions are executed concurrently. The action `mobile_wait_for_completed` can be used in order to synchronize the execution of the aforementioned actions. If the `mobile_wait_for_completed` action is executed, then the execution is blocked until the execution of all currently executed actions of the mobile platform is finished. All other actions are executed synchronously. Thus, the execution of no other action is started until the execution of the currently executed action has finished.

In addition to the actions shown in Table 9.2, the domain model contains a planning operator for the action `do_route(From,To)`. This action is only used in order to determine the expected cost of moving to a certain waypoint, but it is not supposed to be executed by the robot. It is a subtask of the `route(Visited,From,To,Route)` task (see Figure 7.4).

| Action Primitive | Description |
|---|---|
| set_ptu(Pan,Tilt) | This action sets the pan-tilt-unit to (Pan,Tilt). |
| approach_position(X,Y) | The robot approaches the position (X,Y) directly (i.e., without path planning). |
| approach_pose_planned(X,Y,Degree) | The robot approaches the pose (X,Y,Degree) using the path-planner of the Roblet framework (Westhoff et al., 2006). |
| approach_pose_direct(X,Y,Degree) | The robot approaches the pose (X,Y,Degree) directly (i.e., without path planning). |
| move_forward_no_ca(Distance) | The robot moves Distance millimeters forward. The collision avoidance is disabled. |
| mobile_wait_for_completed | If this action is executed, then the execution is blocked until the execution of all currently executed actions of the mobile platform is finished. |
| release_brakes | This action releases the brakes of the mobile platform. |
| apply_brakes | This action applies the brakes of the mobile platform. |
| move_towards(Distance,Degree) | The robot moves Distance millimeters in the direction that is Degree degree relative to the current orientation. |
| rotate_rel(Degree) | The robot rotates Degree degree relative to the current orientation. |
| rotate_abs(Degree) | The robot rotates to the absolute orientation Degree degree. |
| rotate_towards(X,Y) | The robot rotates so that the orientation is in the direction of the point (X,Y). |
| start_trajectory_generator | This action starts the trajectory generator of the arm. |
| stop_trajectory_generator | This action stops the trajectory generator of the arm. |
| open_fingers | This action opens the fingers of the robot hand. |

Continued on next page

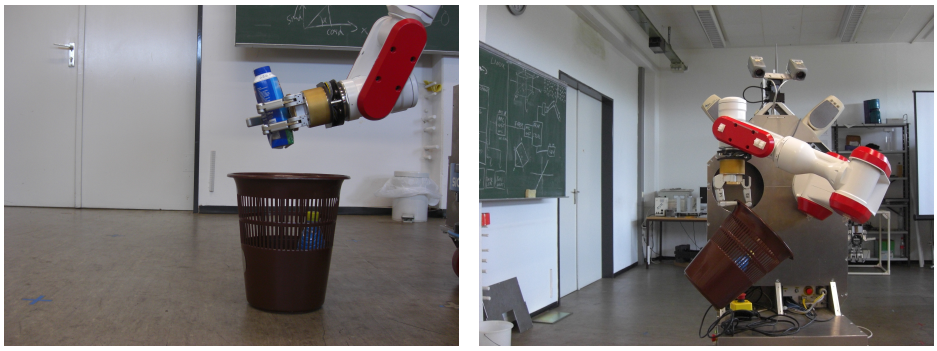
| Action Primitive | Description |
|---|---|
| `close_fingers` | This action closes the fingers until a predefined force is reached or the hand is completely closed. |
| `set_arm_tcp(X,Y,Z,Angle1,Angle2, Angle3,Speed,Mode)` | This action sets the tool-center point to the given position with the speed `Speed` and in the mode `Mode`. The mode can be 99 (ASCII code for 'c') for *cartesian interpolated mode* and 106 (ASCII code for 'j') for the *joint interpolated mode*. See Lloyd and Hayward (1992) for more details of the control of the arm. |
| `set_arm_joints(J1,J2,J3,J4,J5,J6, Speed,Mode)` | This action sets the joints to the given value with the speed `Speed`. The mode can be 99 (ASCII code for 'c') for *cartesian interpolated mode* and 106 (ASCII code for 'j') for the *joint interpolated mode*. See Lloyd and Hayward (1992) for more details of the control of the arm. |
| `sense(percept(laser, query(open(D),I,C),Response))` | This action determines an instance of `open(D)` using the laser scanners. |
| `sense(percept(vision,query(pos(Obj, Pose,X,Y),I,C), Response))` | The action determines the relative position of the object `Obj` lying on a table with respect to the pose `Pose`. It uses the vision system as an external knowledge source. |
| `sense(percept(laser,query(rel_pos( Dist,Degree),I,C),Response))` | The action determines the distance and direction of the nearest object with the laser scanners. |
| `sense(percept(laser,query(rel_pos( X,Y,Degree),I,C),Response))` | The action determines the relative position and direction of the nearest object with the laser scanners. |
| `sense(percept(laser,query( free_ahead(Dist),I,C),Response))` | The action determines with the lasers scanners how many millimeters the robot can move forward without hitting an obstacle. |
| `sense(percept(call,query( at_pose(X,Y,Deg),I,C),Response))` | The action determines the pose of the robot via calling an external component. |

Continued on next page

| Action Primitive | Description |
|---|---|

Table 9.2.: Description of the robot actions that were used for the real-world experiments with the service robot TASER.

### 9.1.3. Experiments



(a) TASER bins a milk package. (b) TASER picks up a garbage can.

Figure 9.1.: Robot actions that were executed during the experiments with the service robot TASER.

This section describes the experiments that were conducted with the physical robot TASER. The conducted experiments are not mainly intended to evaluate a complete robot system, but to evaluate the proposed plan-based control system ACogControl described in Chapter 6. For the real-world experiments, some simplifications (that are not relevant for the main part of this work) were used for the visual perception of objects on a table. Objects on the table are detected with the help of a human instructor by drawing a box around the object of interest in a graphical user interface, as described in Weser (2010). How long acquired information should be stored in the memory is an interesting and highly relevant research question. However, it is out of the scope of this work. It is assumed that all acquired information stays valid during the execution of a single experiment run.

In all experiments, the robot had the task to clean a table. Cleaning a table includes the execution of several typical service robotic tasks including:

- pick up an object from a table,
- navigate to a desired goal position,
- find a garbage can,
- throw away objects,

- and pick up a garbage can.

Initially, the robot had no knowledge about dynamic aspects of the environment including:

- what unknown (i.e., in addition to known objects) dynamic objects exist in the world,
- the number of objects on the table,
- the position of the objects on the table,
- the position of objects on the ground that can obstruct a passage,
- the location of the garbage can,
- between which adjacent waypoints it can actually traverse,
- and the state of the doors.

The initial domain model used for the experiments with the physical robot is illustrated by Figure 7.1. The robot is a priori equipped with a navigation graph, with knowledge about the location and dimensions of tables, and with knowledge about the location of doors. All waypoints of the navigation graph that are printed blue and linked with the table constitute waypoints from which the robot can detect and manipulate objects on the table. As illustrated by Figure 7.1, the robot initially knows a $4\,m^2$ large area in which the garbage can is located. For the purpose of throwing something into or picking up the garbage can, the robot needs to determine a more precise location.

Two sets of experiments were performed with the physical robot. The two sets differ in terms of the number of objects that were located on the table that should be cleaned. For the experiments of the first set, one object was lying on the table, whereas for the experiments of the second set, three objects were on the table. The objective of the conducted experiments is to give answers to the following questions:

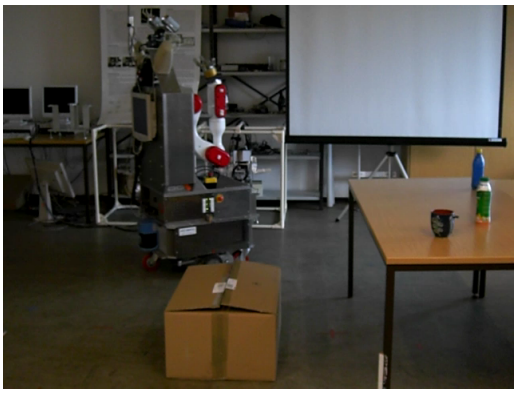
- Is the robot always able to perform the given tasks, although a lot of relevant information is not available a priori?
- How often does the control architecture switch between planning and acting?
- How much planning time is necessary for a planning phase?
- How long is the execution time of an action primitive?
- How does the system behave in a more difficult situation (e.g., with more objects)?

Each set of experiments consists of five runs with varying situations. Beside other aspects, the situations varied in terms of

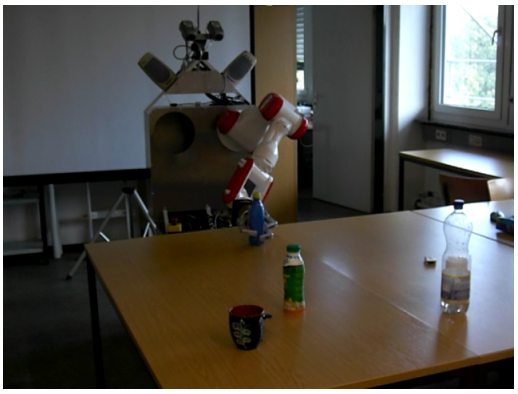
- the type and location of objects that obstruct a passage,
- the type and location of objects on the table,
- the location of the garbage can,
- the start pose of the robot,
- and the state of the doors.

For all conducted experiments, the robot successfully performed the given task. Some of the actions that were performed during an experiment run are shown in Figure 9.2. In all experiment runs, the plan-based controller first realized that the robot needs to determine whether there is an object on the table. In order to do that, TASER autonomously navigated in a cluttered environment from its random start pose to a pose from which it can detect objects and their location on the table (see Figure 9.2a). The autonomous navigation is achieved via continual path planning described in Section 7.1 without using the methods shown in Figure 7.5. If there was an object on the table, then TASER navigated to a position from which it was able to reach the object. Afterwards, it generated a path for the manipulator and picked up the object (see Figure 9.2b). Subsequently, it navigated to a pose where the center of the expected location of the garbage can was in front of it. It determined the exact position of the garbage can with the laser scanner assuming that no other objects were in the expected area of the garbage can. Afterwards, it approached a position from which it could throw the object in the garbage can, planned a path for the manipulator and threw the object away (see Figure 9.2c). This procedure was repeated until no objects were found on the table. If the table was clean, then TASER picked up the garbage can (see Figure 9.2d), navigated to the corridor (see Figure 9.2e and 9.2f), and put the garbage can down. Continual path planning was only used locally inside rooms. In order to navigate to the corridor, TASER first navigated to an open door and then crossed it.

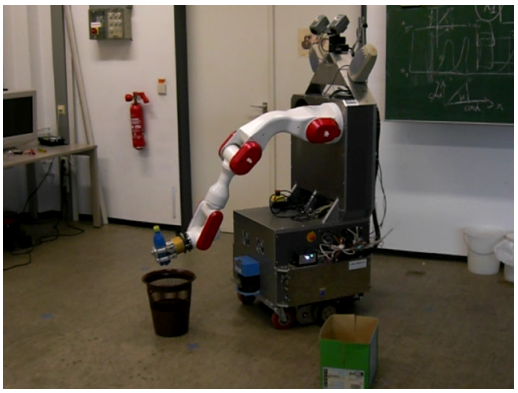
The experimental results for the first set of experiments are shown in Table 9.3. On average, ACogControl divided the overall task of cleaning the table and bringing the garbage out into 38.4 planning and execution phases, executed 173 action primitives for an experiment run, and planned 4.524 steps (i.e., action primitives) ahead. Planning only several steps ahead in situations where it is reasonable—or necessary—to acquire additional information during execution is an important property of the proposed system. It divides the overall planning problem, so to speak, into a set of smaller planning problems. During the experiments, the robot acquired information via 22.6 primitive percepts (i.e., sensing actions) and 29.2 high-level percepts (i.e., percepts that result from a multimodal integration process); which means 1.34 percepts per planning phase on average. The CPU time used by the



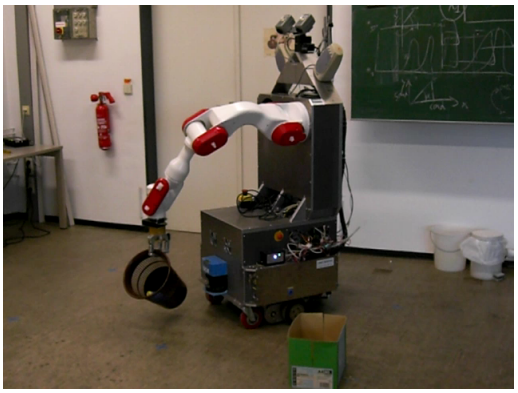
(a) TASER tries to find a pose from which it can pick up the object.



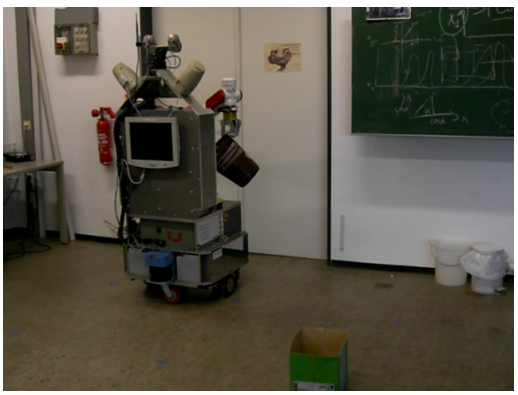
(b) TASER picks up the object from the table.



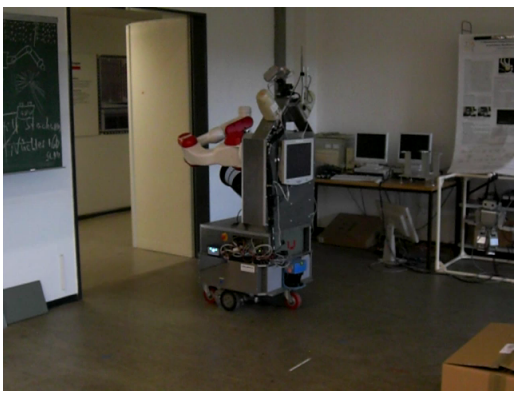
(c) TASER bins the object.



(d) TASER picks up a garbage can.



(e) TASER determines that the first door is closed.



(f) TASER determines that the second door is open and moves to the corridor via that door.

Figure 9.2.: Illustration of the actions that have been performed by the service robot TASER during an experiment run.

| Name | aver. | min | max |
|---|---|---|---|
| planning/execution phases | 38.4 | 33 | 45 |
| action primitives per phase | 4.524 | 1 | 25 |
| action primitives per run | 173 | 157 | 194 |
| percepts | 51.8 | 43 | 65 |
| primitive percepts | 22.6 | 19 | 27 |
| high-level percepts | 29.2 | 24 | 38 |
| percepts per phase | 1.34 | 1.3 | 1.44 |
| planning CPU time per action | 0.0066 s | – | – |
| planning CPU time per phase | 0.0295 s | 0.0005 s | 0.2060 s |
| planning CPU time per run | 1.1550 s | 0.8106 s | 1.8530 s |
| execution time – action | 3.417 s | 0.009 s | 41.983 s |
| execution time – run | 594 s | 471 s | 701 s |

Table 9.3.: Experimental results for the first set of experiments with the physical service robot TASER.

plan-based controller (including planning and reasoning) is 0.0295 seconds on average and maximally 0.2060 seconds per planning phase. For the complete execution of the given tasks, 1.155 seconds CPU time on average is used for planning and reasoning. This is very low compared to the mean execution time, which is 3.417 seconds for an action primitive and 594 seconds for a complete experiment run. Thus, the ratio of time used for planning and reasoning to the overall execution time is very low at $\frac{1.155\,\text{s}}{594\,\text{s}+1.155\,\text{s}} = 0.0026$.

For the second set of experiments, three objects are located on the table. The results are shown in Table 9.4. The fact that two additional objects are on the table makes the task more extensive. Thus, TASER needed to acquire more information (e.g., the location of the additional objects) and execute more action primitives. The fact that more information needs to be acquired leads to more planning and execution phases, since the underlying planning algorithm terminates a planning phase whenever it decides to acquire additional information (see Chapter 5). As shown in Table 9.4, the overall tasks were on average decomposed into 62 planning and execution phases. On average, 313 action primitives were executed during each run, and additional information was acquired via 33 primitive and 36 high-level percepts.

Some information (e.g., whether a free path exists between connected waypoints) is only acquired once, even when the same task (e.g., move from the table to the garbage can) is executed multiple times. For example, when the robot navigates the first time from the table to the garbage can, then it does not have any information about objects that obstruct a passage. This information is acquired while navigating to the garbage can. However, when the robot has to navigate from the table to the

| Name | aver. | min | max |
|---|---|---|---|
| planning/execution phases | 62 | 56 | 73 |
| action primitives per phase | 5.14 | 1 | 27 |
| action primitives per run | 313 | 276 | 359 |
| percepts | 69 | 58 | 78 |
| primitive percepts | 33 | 30 | 36 |
| high-level percepts | 36 | 28 | 42 |
| percepts per phase | 1.13 | 1.04 | 1.20 |
| planning CPU time per action | 0.0062 s | – | – |
| planning CPU time per phase | 0.0317 s | 0.0005 s | 0.1898 s |
| planning CPU time per run | 1.9334 s | 1.5396 s | 2.2994 s |
| execution time – action | 3.363 s | 0.008 s | 34.17 s |
| execution time – run | 1056 s | 897 s | 1259 s |

Table 9.4.: Experimental results for the second set of experiments with the physical service robot TASER.

garbage can again for the purpose of throwing the second object into the garbage can, then it has already acquired considerable information about its environment. Therefore, less information needs to be acquired, and the planner can plan more steps ahead. Technically speaking, for the second set of experiments, the proposed control system planned 5.14 steps ahead on average and with 1.13 has a significantly lower percepts per phase ratio. Acquired information remains in the knowledge base until it is invalidated by the effects of the robot's actions. The fact that acquired information can also be invalidated by external events is not considered (cf. Section 4.5). Another interesting observation is the fact that, compared to the first set of experiments, the average CPU time of a planning phase increases to 0.0317 seconds, but the maximum CPU time is even lower with 0.1898 seconds. On average, a complete experiment run needed 1.9334 seconds for planning and 1056 seconds for execution. Thus, we have an even lower planning to overall execution time ratio of $\frac{1.9334\,\mathrm{s}}{1.9334\,\mathrm{s}+1056\,\mathrm{s}} = 0.0018$ than for the first set of experiments.

In summary, we can say that the proposed control architecture can enable a robot to perform typical service robotic tasks even if no information about the dynamic aspects of the environment is initially available. The fact that the overall planning problem is automatically divided into a set of smaller planning problems makes the approach sufficiently fast for the described scenarios.

### 9.1.4. Example Plans

This sections shows and describes a sequence of three subsequent plans for the purpose of giving the reader a better impression of the nature of generated plans.

```
clean(table1) [...]
└─ clean_obj(blue_bottle) [...]
   ├─ pick_up(blue_bottle) [...]
   │  ├─ navigate(pose(left(table1),deg(0))) [5.81 s]
   │  │  ├─ ensure_drive_pose [0 s]
   │  │  ├─ go_on_roadmap(w15) [0 s]
   │  │  ├─ navigate([w15],w15,left(table1)) [5.77 s]
   │  │  │  └─ approach(left(table1)) [5.77 s]
   │  │  │     ├─ <approach_position(9168,14536)> [0.04 s]
   │  │  │     └─ <mobile_wait_for_completed> [5.73 s]
   │  │  └─ <rotate_abs(0)> [0.016 s]
   │  └─ grab_from(blue_bottle,Pose) [...]
   ├─ navigate_to_entity(garbage_can) [...]
   └─ bin(blue_bottle) [...]
```

Figure 9.3.: Partial plan for the task `clean(table1)`.

The presented plans were generated during an experiment run of the experiments described Section 9.1.3.

Figure 9.3 shows a partial plan for the task of cleaning table `table1`. The fact that only a partial plan exists for a task is illustrated by a subsequent "`[...]`". In contrast, if the plan contains a complete plan for a task, then the execution time of the task is shown after the term that represents the task. For example, the task of navigating with the mobile platform to the pose `pose(left(table1),deg(0))` was executed by TASER in 5.81 seconds (see Figure 9.3). The plan shown in Figure 9.3 is incomplete. The planner was not able to further decompose the task `grab_from(blue_bottle,Pose)`, because the location of the object `blue_bottle` was not available. Therefore, the planner stopped the planning process.

Subsequently, the plan-based controller integrated the execution of a knowledge acquisition task that determines the relative position of the object. In this case, the planner generated a complete plan for this task. The generated plan is shown in Figure 9.4. The plan was executed in 18.342 seconds. As specified by the plan, the robot first moved its arm aside so that it had a free view of the table and then used the vision system for the purpose of determining the relative position of the object.

Afterwards, the controller instructed the planner to continue to plan the overall task of cleaning the table `table1`. Based on the new information about the position of the object `blue_bottle`, the planner generated a complete plan for the task of picking up the object. This plan is shown in Figure 9.5. For clarity reasons, 8 primitive and 14 non-primitive tasks are not explicitly show in Figure 9.5. They are hinted at by syntactical constructs of the following form:

```
...<< N subtasks | M primitives >>
```

```
det(vision,pos(blue_bottle,pose(left(table1),deg(0),X,Y),[],[],
pos(blue_bottle,pose(left(table1),deg(0)),734,51))) [18.342 s]
├─ navigate(pose(left(table1),deg(0))) [0 s]
├─ move_manipulator(joints(0,-61,60,0,20,0)) [6.996 s]
│  ├─ <start_trajectory_generator> [0.099 s]
│  ├─ move_manipulator([joints(-90.0,-61.0,152.0,0.0,20.0,-90.0)],
│  │  joints(0,-61,60,0,20,0)) [6.870 s]
│  │  └─ direct_move_manipulator(modus(0.5,106), joints(0,-61,60,0,20,0)) [6.870 s]
│  │     └─ <set_arm_joints(0,-61,60,0,20,0,0.5,106)> [6.870 s]
│  └─ <stop_trajectory_generator> [0.026 s]
└─ <sense(vision,pos(blue_bottle),pos(734,51))> [11.346 s]
```

Figure 9.4.: Plan for the task of determining the relative position of the blue bottle from the waypoint `left(table1)`.

For such a construct, `N` indicates the total number of subtasks, and `M` indicates how many of these tasks are primitive.

The robot picked up the object in 46.59 seconds. However, not all necessary information (e.g., the position of the garbage can) was available in order to generate a complete plan for the task of navigating to the garbage can. Thus, an additional knowledge acquisition task needed to be integrated and performed in the subsequent planning phase.

## 9.2. Simulation-Based Evaluation

In addition to the experiments with the physical robot TASER presented in Section 9.1, a set of simulation-based experiments was performed. The simulation-based experiments use the same domain model—and extensions thereof—that was used for the experiments with the physical service robot TASER as well as three well-known domains from the AI planning community.

Two additional sets of experiments were conducted for the purpose of getting deeper insights with respect to the following questions:

- How does the system scale to more complex domains?
- How does the amount of initial knowledge affect the performance of the proposed system?

The simulation-based experiments where conducted on a 64-bit Intel Core 2 Quad Q9400 with 4 GB memory and SWI-Prolog 6.0.0 as the underlying Prolog engine. The system only used one processor.


Figure 9.5.: Plan for the task of determining the relative position of the blue bottle from the waypoint `left(table1)`.

### 9.2.1. ACogSim

Providing a simulation environment for the evaluation of a continual planning system is not a trivial task (Brenner and Nebel, 2009). It is usually more difficult than the evaluation of classical planning systems, since it is additionally necessary to simulate the execution of actions.

In the context of this work, a simulator, namely *ACogSim*, for the environment has been developed. ACogSim enables the systematical evaluation of the whole plan-based control architecture—including execution—described in Chapter 6. The ACogSim simulator works similar to MAPSIM, as described by Brenner and Nebel (2009). In contrast to the agent, ACogSim has a complete model of the domain. When the executor executes an action, then the action is sent to ACogSim. ACogSim checks the preconditions of actions at runtime prior to the execution and updates its simulation model according to the effect of the actions. In this way, ACogSim simulates the execution of actions and guarantees that the executed plans are correct.

The outcome of sensing actions is also simulated by ACogSim. Let $D_{Msim}$ be the (complete) domain model of an ACogSim instance. Let $\texttt{sense}(l, I, C, ks, l_r)$ be a sensing action for which the parameters have the same meaning as for a knowledge acquisition task (see Definition 4.7). The result of a sensing action $\texttt{sense}(l, I, C, ks, l_r)$ is

- an additional instance $l\sigma$ of $l$ if such an instance can be derived with respect to $D_{Msim}$,
- *impossible* if it can be derived that the existence of an additional instance of $l$ is impossible,
- or *indeterminable* otherwise.

As a proof of concept, all experiments with the physical robot TASER (see Section 9.1) have been rerun with the ACogSim simulator. In the simulation-based environment, the ACogControl system planned and executed exactly the same plans as in the real-world environment. This demonstrates that the simulator is detailed enough to confront the proposed control system with identical situations, although ACogSim does not simulate the physical action primitives in detail.

The simulator is encapsulated in a corresponding executor that provides the same interface as the executor for the physical robot TASER. Therefore, the proposed control system does not have to be modified in order to work with the simulator. Whether the simulator or the physical robot executes the actions is not visible for the control system.

### 9.2.2. Service Robotic Domain

The domain model (and extensions thereof) that was used for the real-world experiments (see Section 9.1) was also used for the experiments with the ACogSim simulator.

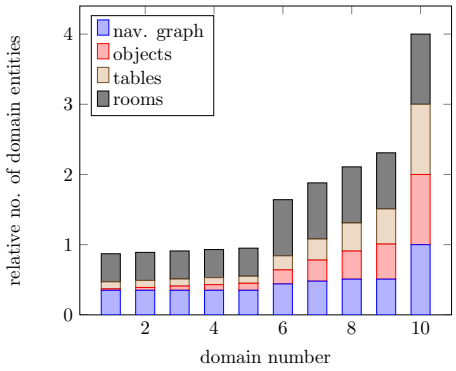



Figure 9.6.: Number of domain model entities relative to the number of entities of the same type for the largest domain (Domain 10).

### Increasingly Complex Service Robotic Domain

For the purpose of evaluating how the system scales to larger domains, the domain model used for the experiments with the physical service robot TASER (see Section 9.1) is successively extended. The smallest domain (Domain 1) is identical with the domain used for the first set of experiments with the physical robot. It contains two rooms, a navigation graph with 23 vertices and 39 edges, one table, and one object on a table (see Figure 7.1). The largest domain (Domain 10) contains five rooms, a navigation graph with 69 vertices and 180 edges, 10 tables, and 50 small objects on the tables. How the number of entities of the domain model is successively increased is shown by Figure 9.6. Figure 9.6 shows the number of domain entities for each type (i.e., rooms, tables, navigation graph elements and objects on the table) relative to their number in the largest domain (i.e., Domain 10). The amount of domain entities is presented relative to their amount in the largest domain in order to make relative changes to their amount better visible. For example, 0.5 tables in Figure 9.6 correspond to $0.5 \times 10 = 5$ table entities in the domain model.

The domain model is extended in three stages. In the first stage (Domain 1 - Domain 5), each following domain is enlarged by one object that needs to be thrown away. Second, from Domain 6 to Domain 9, subsequent domains are enlarged by one table, five objects, and a couple of waypoints. Furthermore, human-robot interaction is added as an additional knowledge source, and the additional constraint `garbage(Object)` is added to the precondition of the HTN methods that define how to clean a table. Hence, the robot is only allowed to bin an object if it knows that the object is considered as garbage. If this information is not available, then the robot can acquire it by means of picking up the object, navigating to the office room, and asking the person in the office whether the object should be thrown away. In the last stage, an additional room with 5 tables and 25 objects is added, and the size of the navigation graph is almost doubled. The largest domain model (Domain 10) is illustrated by Figure 9.7.

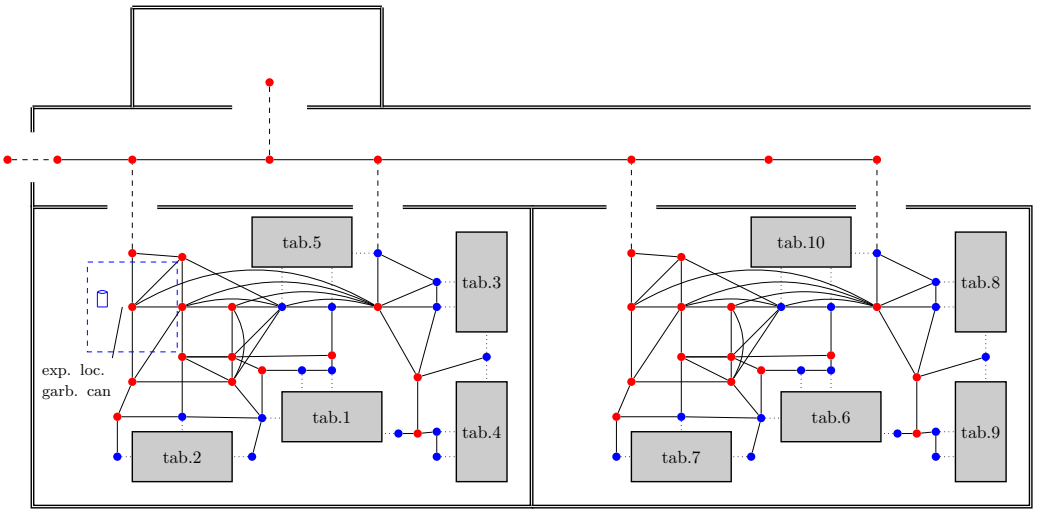



Figure 9.7.: Illustration of the most complex domain model 10.

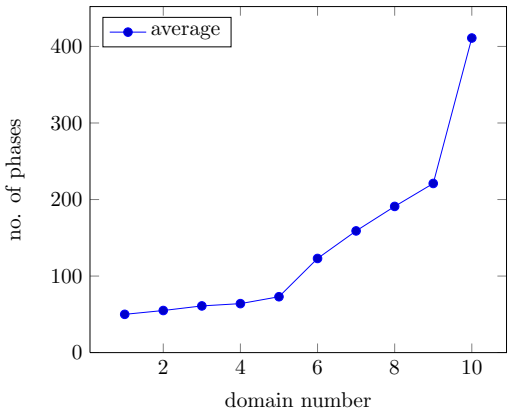
The proposed plan-based control architecture successfully performed the given task for all conducted experiment runs. The detailed results of the experiments are show in Figure 9.8a - 9.8e. The number of planning and execution phases and the number of percepts increases approximately linearly to the overall domain size (see Figure 9.8a - 9.8b). Figure 9.8c shows that the average and maximum size of the generated plans increase slightly for more complex domains. The average CPU time required for a phase (see Figure 9.8d) does not exceed 0.1 seconds, and the maximum CPU time required for a phase stays under 0.4 seconds. This can be considered as sufficiently fast for a service robot domain. In particular, the time needed for a planning phase is—even for the largest domain—quite low compared to the average execution time of an action primitive, which is approximately 3.4 seconds (see Table 9.3 and 9.4). Figure 9.8e shows that, even for the largest domain, the time that is necessary for planning a single primitive action stays around 6 milliseconds.

Summing up, the conducted experiments indicate that the proposed control architecture scales comparably well to an increasingly complex service robot domain. The main reason for this characteristics is the fact that the continual planner automatically partitions the overall planning problem into a set of simpler planning problems and therefore seems to be less affected by the domain size than classical planning approaches.

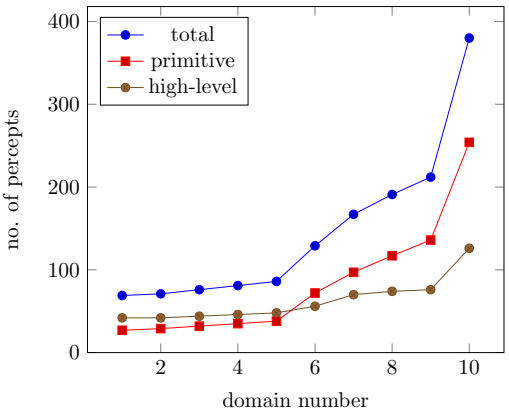
### Performing Tasks with an Increasing Amount of Initial Knowledge

10 additional set of experiments were performed for the purpose of dealing with the following question:

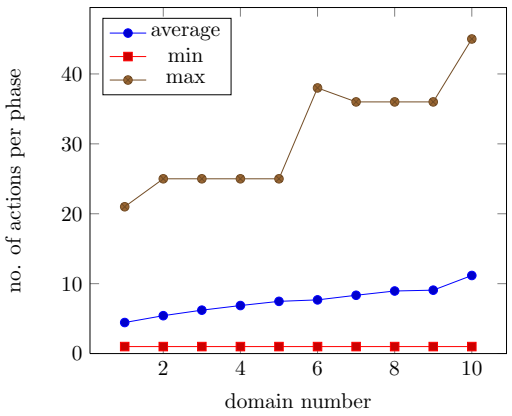
- How does the performance of the system relate to the amount of information that is initially available?



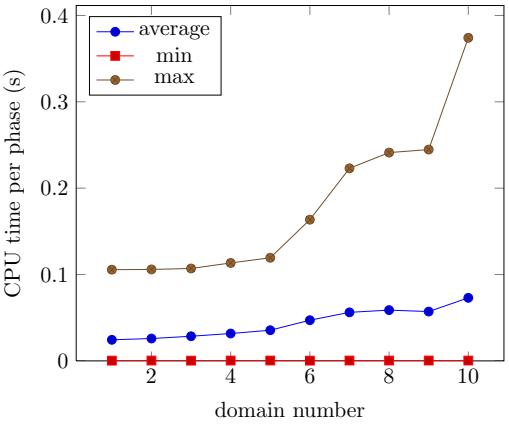

(a) Number of planning and execution phases.



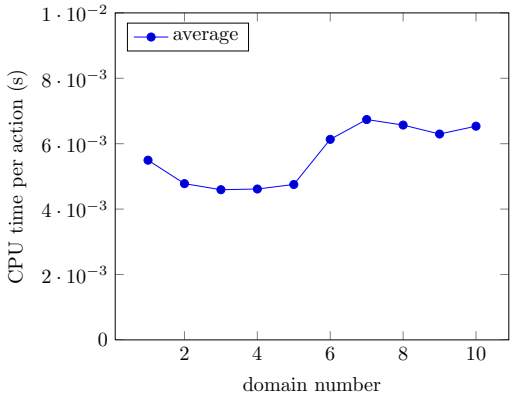

(b) Number of percepts.




(c) Number of action primitives per phase.

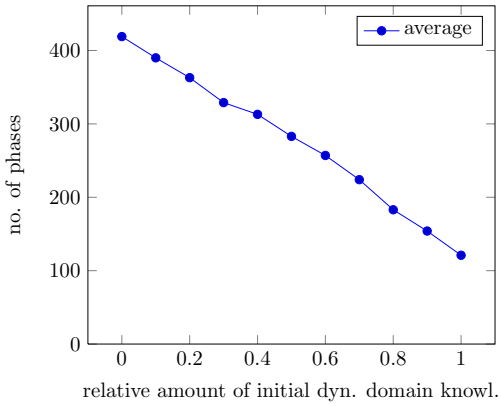



(d) CPU time per phase.

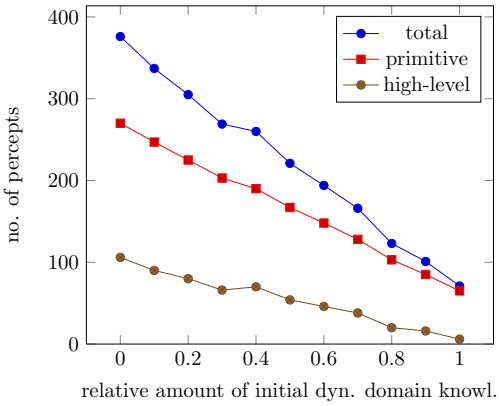



(e) CPU time per action.

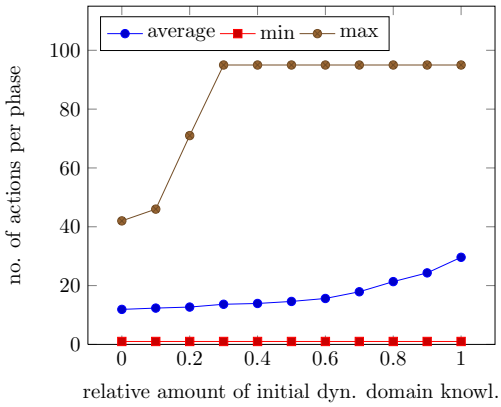
Figure 9.8.: System behavior for an increasingly complex service robotic domain.



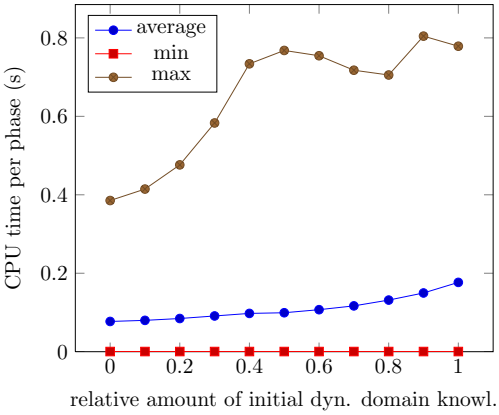

(a) Number of planning and execution phases.



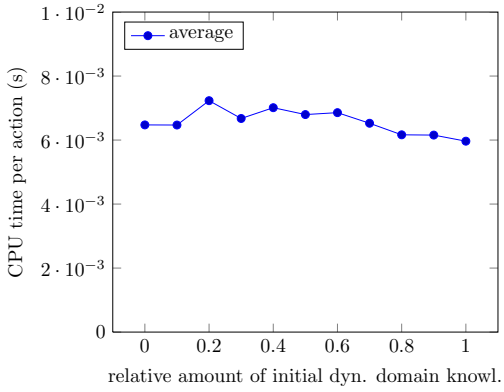

(b) Number of percepts.




(c) Number of action primitives per phase.




(d) CPU time per phase.




(e) CPU time per action.

Figure 9.9.: System behavior for an increasing amount of initial knowledge about dynamic aspects of the environment for the simulated service robotic domain.

In order to deal with this question, information about the dynamic aspects of the environment (e.g., the exact location of objects, the state of doors or the actual connectivity between waypoints) was randomly added in 10 steps of equal size (i.e., with an equal amount of added facts) to the largest domain model (Domain Model 10) described in Section 9.2.2.

The results of the experiments are shown in Figure 9.9a – 9.9e. The number of planning and acting phases as well as the number of percepts decreases with an increasing amount of initially available knowledge (see Figure 9.9a and 9.9b). In contrast, the average and maximum number of action primitives that are executed during a phase increases, since the planner has more information and thus can plan more steps ahead (see Figure 9.9c). Figure 9.9d shows that the average CPU time per phase increases with an increasing amount of initial knowledge. This is mainly due to the increase of the average number of action primitives per phase, since the average CPU time needed for planning a single primitive action is not significantly affected by the amount of initial knowledge (see Figure 9.9e). Looking at this result from the opposite direction results in a quite interesting perspective, since it indicates that a lot of missing information does not make the problem computationally harder. In contrast, the results indicate that if the proposed control architecture has less initially available information, then it partitions the overall process into a larger sequence of smaller planning and execution phases. In this way, the CPU time required for a planning phase decreases with a decreasing amount of initial knowledge.

## 9.2.3. Additional Planning Domains

The studies described in Section 9.2.2 were additionally conducted with three well-known AI planning domains. For these studies the *depots* and *rovers* domains from the third international planning competition (see Long and Fox (2003)) as well as the well known *blocks world* (Winograd, 1972) domain were used. The domain model specifications were adapted to the domain model described in Chapter 3 and 4. The basic specifications of the domain models can be found in Appendix B.

### Increasingly Complex Domains

Five increasingly large domain model instances of the *rovers*, the *depots* and the *blocks world* domain were used to determine how the proposed system deals with an increasingly complex domain. How many facts in the form of ground literals the domain model instances contained is shown in Table 9.5. For example, the smallest domain model instance of the rovers domain contained 47 facts, whereas the largest domain instance contained 1758 facts.

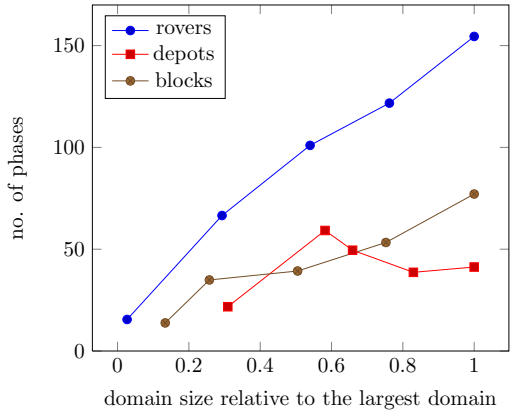
All domain instances initially contained all information that is necessary in order to generate a complete plan. 100 experiment runs were performed for each domain model instance. For each experiment run, 50 percent of the facts were randomly removed from the corresponding domain model instance so that often no complete

| Domain | Instance | No. of facts |
|---|---|---|
| Rovers | 1 | 47 |
| | 2 | 515 |
| | 3 | 949 |
| | 4 | 1340 |
| | 5 | 1758 |
| Depots | 1 | 272 |
| | 2 | 512 |
| | 3 | 580 |
| | 4 | 730 |
| | 5 | 880 |
| Blocks World | 1 | 270 |
| | 2 | 520 |
| | 3 | 1020 |
| | 4 | 1520 |
| | 5 | 2020 |

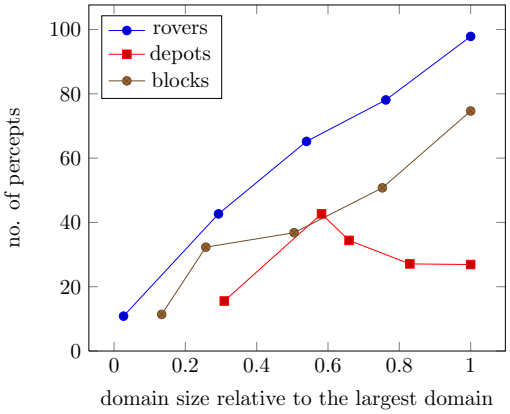
Table 9.5.: Number of facts for the five increasingly large domain model instances of the *rovers*, *depots*, and *blocks world* domain.

plans were found in the fist planning phase and the simulated agent needed to acquire additional information in order to perform a given task. For all runs, the same task that should be performed was used for each domain. The task for the rovers domain was to get soil data from a certain waypoint. Delivering a crate to a given depot was the task for the depots domain, and moving a block onto another block was used for the blocks world domain.

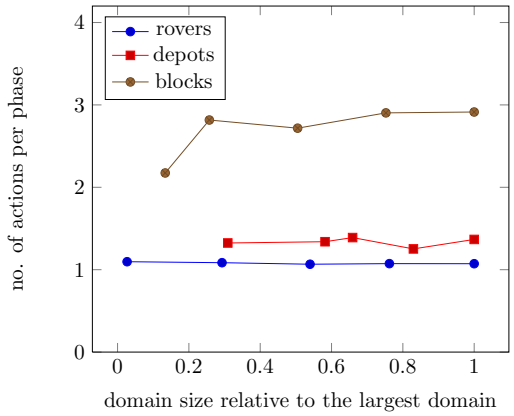
The experimental results are shown in Figure 9.10. Figure 9.10a shows that the number of planning and execution phases scales—like for the service robotic domain—linearly to the domain size for the rovers and the blocks world domain. The depots domain, however, shows a different trend. The number of phases increases from the first to the second domain instance, slightly decreases from the second to the fourth domain, and finally goes a bit up again. One reason for this behavior is that the task used for the depots domain does, in contrast to the tasks used for the rovers and blocks world domain, not necessarily become more difficult in terms of the actions that have to be performed and the amount of necessary knowledge. To put it another way, the amount of knowledge that is necessary in order to move a crate



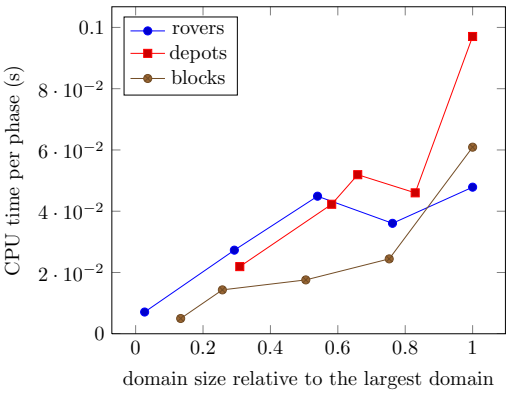

(a) Number of planning and execution phases.



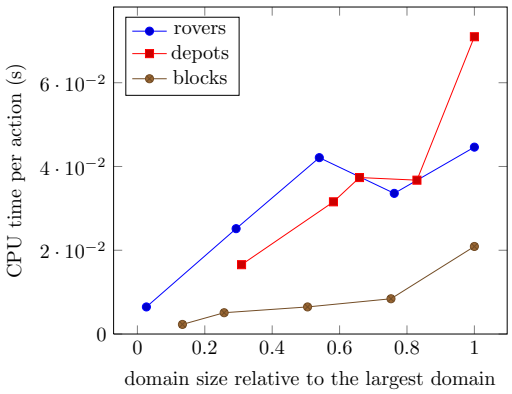

(b) Number of percepts.




(c) Number of action primitives per phase.




(d) CPU time per phase.




(e) CPU time per action.

Figure 9.10.: System behavior for an increasingly complex domain for the *rovers*, *depots* and *blocks world* domains.

to a certain depot is not necessarily larger for the larger domain instances. Thus, the control system does not necessarily need to add more planning and execution phases in order to acquire more information. Like for the service robotic domain, Figure 9.10b approves that the number of percepts behaves almost identical as the number of phases. The average number of primitive actions that are planned and executed during one planning respectively execution phase is shown in Figure 9.10c. The number of primitives per phase approximately increases 25 percent from the first to the second domain instance of the blocks worlds domain and then stays almost constant. For the rovers and the depots domain, the increasingly complex domain instances do not significantly affect the number of action primitives per phase. The plots shown in Figure 9.10d and 9.10e look very similar, since the number of primitives per phase stays almost constant for all domains. Moreover, these figures approve that the plan-based control architecture scales well to more complex domains compared to classical planning approaches that usually show an exponential behavior.

### Increasing Amount of Initial Knowledge

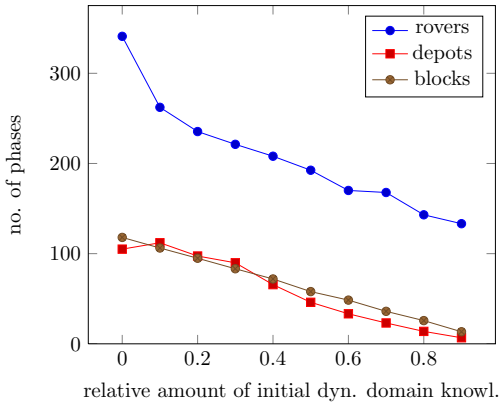
The rovers, depots and blocks world domain were also used as a testbed for the second simulation-based study already conducted with the service robotic domain (see Section 9.2.2). The objective of this study is to determine how the ACogControl system behaves if the agent is equipped with more prior knowledge. For this study, the largest instances of the rovers, depots and the blocks world domain models are used. Thus, the complete domain model instance that includes all necessary facts contains 1758 facts for the rovers, 880 facts for depots, and 2020 facts for the blocks world domain.

The results presented in this section differ from the previously published results in Off and Zhang (2012), since the algorithm of the controller is modified. In contrast to the algorithm used for the results described in Off and Zhang (2012), the current version of the controller causes replanning whenever additional information has been acquired so that it does not get lost in a local minima (see Section 6.2). Therefore, the current version of the controller usually needs less planning and execution phases.

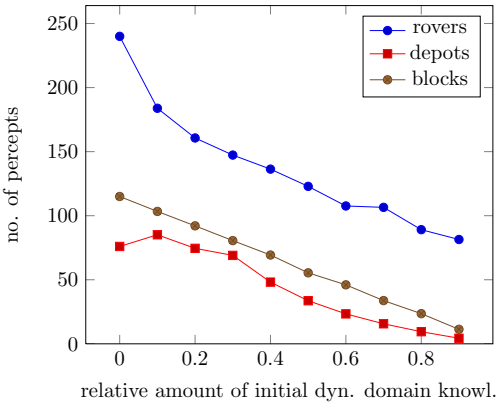
10 experiments were conducted for all domains with 100 runs per experiment, except for the first experiment where 1 run was sufficient. Let $\mathfrak{f}_{all}$ be the number of facts of a domain model, then $\frac{11-i}{10}\mathfrak{f}_{all}$ facts were removed in all runs of the $i$-th experiment from the domain model of the agent. Hence, in the first experiment all facts were removed (for each domain) from the agent's domain model. For the first experiment only one run was performed, since there is only one way of removing all facts from the domain model.

ACogControl was able to correctly perform the given task for all domains and all runs—even in situations where all facts were removed from the domain model of the agent. The more detailed results of the experiments are shown in Figure 9.11.

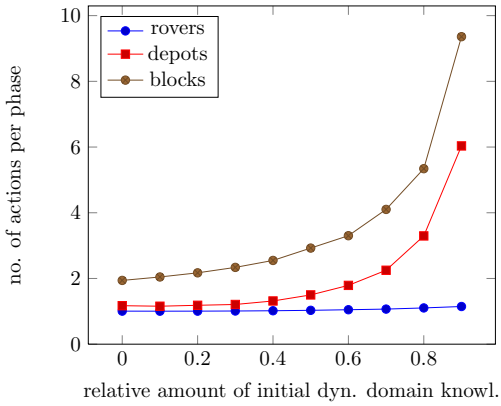
The average number of necessary planning and execution phases is show in Figure 9.11a. The average number of planning and execution phases decreases with an



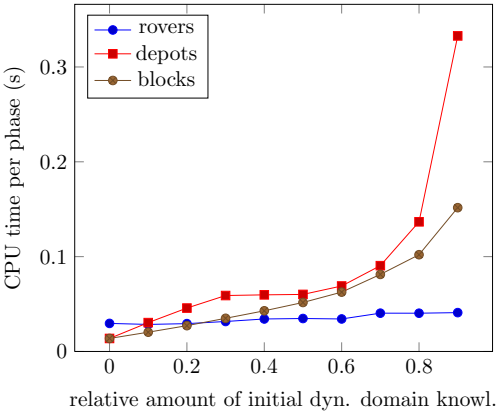

(a) Number of planning and execution phases.



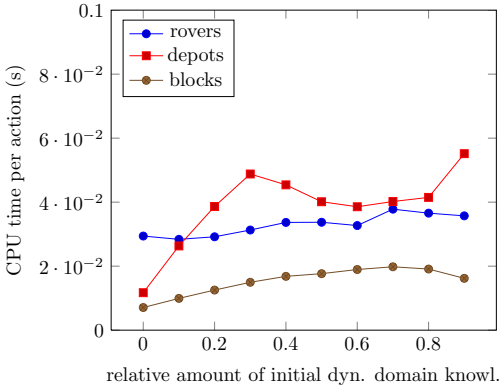

(b) Number of percepts.




(c) Number of action primitives per phase.




(d) CPU time per phase.




(e) CPU time per action.

Figure 9.11.: System behavior for an increasing amount of initial knowledge about dynamic aspects of the environment for the three additional planning domains.

increasing number of initial information, since the agent needs to stop the planning process and execute knowledge acquisition activities less often for situations where it has more prior knowledge. The plot of the number of percepts is again very similar to the plot of the number of phases (see Figure 9.11b). Figure 9.11c shows that the number of planned and executed action primitives per phase expeditiously increases with more initially available information for the depots and blocks world domain. This behavior seems to be intuitively reasonable, since the planner can plan more steps ahead if more information is available. However, the rovers domain shows a different behavior. For the rovers domain, the plan-based controller often only plans one step ahead even if 90 percent of the dynamic domain knowledge is initially available. The reason for this behavior is that the planner often decides to acquire additional information even if a complete plan can be generated. For the rovers domain the planner spends most of the time for path planning. The domain model contains—like for the service robotic domain—a navigation graph that defines from which waypoint the rover can directly traverse to another waypoint. Instead of trying to generate a complete path, the planner usually chooses the next waypoint and then stops the planning process, because there could be a connection between the next waypoint and another waypoint (e.g., the goal waypoint) of which the planner currently has no information. Thus, it is often more reasonable—according to the underlying cost model—to choose (i.e, plan) the next waypoint, execute the plan, and then determine to which waypoints the next waypoint is connected by means of active knowledge acquisition. Based on the acquired information about adjacent waypoints, the planner now can make a possibly better planning decision. The CPU time for a planning phase quickly increases for the depots and blocks world domain and stays constant for the rovers domain (see Figure 9.11d). However, the fact that the average planning time necessary for a single primitive action stays almost constant for the rovers and blocks world domain, and is not increasing very fast for the depots domain (see Figure 9.11e) indicates that the increase of the CPU time for a phase is mainly due to the increase of the number of primitives that are planned for a phase.

## 9.3. Summary and Conclusion

In this chapter, a couple of real-world and simulation-based experiments have been described that were performed in order to evaluate the proposed plan-based control system ACogControl.

The results of the experiments with the service robot TASER demonstrate that ACogControl can enable a physical service robot to autonomously perform high-level tasks in an unstructured environment even if a lot of relevant information is not initially available, and thus the robot only has an incomplete, open-ended model of its environment. The CPU time necessary for planning is very low compared to the overall execution time. For example, an average experiment run for the first set of experiments with TASER took approximately ten minutes, whereas the required

CPU time for planning was always below two seconds (see Table 9.3).

Planning in open-ended domains is known to be more difficult than planning based on the assumption that all information is available at planning time (Baral et al., 2000). Nevertheless, the experimental results indicate that the proposed approach scales, compared to previous planning approaches, surprisingly well to more complex domain models (see Section 9.2.2 and 9.2.3). The main reason for this result is the fact that the plan-based control system automatically partitions the overall planning process into a set of simpler planning problems. Due to a lack of knowledge, it is not necessarily possible to plan more steps ahead for a larger domain model. Thus, the planning problems do not necessarily become more difficult. If the planning problems do not become more difficult for more complex domains, then usually the set of planning phases increases. In other words, a more complex domain often results in a larger set of (at most) slightly more difficult planning problems. In contrast, for most of the previous planning approaches that assume that planning is monolithic, an increasingly large domain results in a single, increasingly complex planning problem that usually scales exponentially to the size of the domain model. The fact that the proposed plan-based control system generates and executes plans incrementally comes at the cost that the corresponding agent starts the execution of a partial plan, for which it cannot be guaranteed that it is the prefix of a valid plan. However, in the real-world, open-ended environments this work is interested in, it is often impossible to generate a complete plan in advance. Thus, often the alternative to executing a possibly incorrect, partial plan is to not be able to perform a task at all. The main motivation of the proposed control system was to be able to deal with these situations, where it is impossible, or at least unreasonable, to generate a complete plan for a given task prior to executing any action.

The experiments with an increasing amount of initial knowledge (see Section 9.2.2 and 9.2.3) evaluate how the extent of initially available knowledge relates to the performance of the described control system. The experimental results indicate that if the control system initially has access to more knowledge about the state of the environment, then the overall planning problem is partitioned into less, but increasingly difficult planning problems. For the service robot domain, the number of planning phases decreases in the same way as the CPU time for a planning phase increases so that the overall planning time approximately stays constant. In contrast, for the additional planning domains from the AI planning community, the CPU time for a single planning phase increases faster than the number of planning phases decreases so that the overall planning time slightly increases for situations where more knowledge is initially available. Another conclusion that can be drawn from the experimental results is that it is not a problem with respect to the computational complexity if a lot of relevant information is not initially available. More missing information makes the overall execution of a task more difficult, but the computational complexity of the overall planning process is often even reduced.

All the tested domains provide additional insights into the performance characteristics of the described plan-based control system. However, the domain model for

the service robot is the most realistic domain model. It provides more insights with respect to the performance of the system in the context of a complex, real-world situation.

Chapter 10

# Summary and Conclusion

*It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment. When I have clarified and exhausted a subject, then I turn away from it, in order to go into darkness again. The never-satisfied man is so strange; if he has completed a structure, then it is not in order to dwell in it peacefully, but in order to begin another. I imagine the world conqueror must feel thus, who, after one kingdom is scarcely conquered, stretches out his arms for others.* (Carl Friedrich Gauss, 1777 -1855)

## Contents



This chapter summarizes the main results and contributions of this work and outlines possible directions for future work.

## 10.1. Summary

In this work, a novel plan-based control system has been developed. The development is motivated by the fact that artificial agents are supposed to autonomously perform tasks in real-world environments, in spite of being equipped with models of the world they inhabit that are inherently incomplete and open-ended. Most

related approaches have been identified to be inadequate, since they rely on a complete model of the relevant slice of reality and assume that a plan can be generated in a single, monolithic process; while typically neither of these assumptions holds in the light of the complexity and unstructuredness of real-world environments. Comparatively less attention has been paid to task planning and plan-based control approaches that make use of open-ended domain models and can more flexibly interleave planning and execution to deal with the open-endedness of real-world situations.

This work has presented a novel plan-based control system that extends previous approaches so that tasks can be automatically performed in more realistic situations where a lot of relevant information initially is not available. The proposed approach provides the planning and reasoning capabilities to perform tasks in a significantly larger set of situations. The proposed control system is *domain-configurable* (Nau, 2007); thus, the core planning, reasoning and controlling engines are domain independent, but can exploit domain specific information. The proposed control architecture can be viewed as a general, plan-based, high-level control framework. It features the automatic as well as manual integration of external components.

In a nutshell, the main contributions of this work are:

1. **Developing a domain model and reasoning system for open-ended states**. Chapter 3 and Chapter 4 introduced a novel, open-ended domain model for HTN planning systems. This domain model provides the basic knowledge representation and reasoning capabilities. In contrast to the underlying domain model of most of the previous task planning approaches, the proposed domain model can represent open-ended state models. In other words, the domain model can natively deal with incomplete information. It is capable of reasoning about *possible*, *relevant*, and *acquirable* extensions of a state model.

2. **Developing a principled new HTN planning approach for open-ended domains**. Chapter 5 introduced a principled new HTN planning approach. The proposed HTN planning system ACogPlan extends existing approaches in various ways for the purpose of dealing with open-ended state models. ACogPlan additionally considers decompositions that are only applicable with respect to a consistent extension of the (open-ended) domain model at hand. Planning is not assumed to be a monolithic process. The planner automatically decides when it is more reasonable to leave parts of the hierarchical plan unexpanded and acquire additional information prior to making the next planning decision for the task at hand. It was demonstrated that the domain specific information encoded in the domain model of an HTN planner not only helps to more efficiently solve classical planning problems, but can efficiently be exploited in open-ended situations, where a planner additionally has to consider unassured ways to perform a task.

3. **Developing a plan-based control architecture that continually interleaves planning and execution.** Chapter 6 described how the proposed planning system ACogPlan is integrated into the plan-based control system ACogControl. ACogControl is capable of autonomously performing tasks in open-ended domains where a lot of relevant information is not a priori available. Whenever it is more reasonable, or even necessary, to acquire additional information prior to making the next planning decision, the planner defers the overall planning process, the control system automatically creates appropriate knowledge acquisition tasks, and integrates their execution into the overall planning and execution process.

The proposed plan-based control architecture ACogControl was applied to the real-world service robot TASER (see Chapter 8). Real-world and simulation-based experiments indicated that ACogControl can enable a physical service robot to successfully perform tasks in a real-world environment, though a lot of relevant information was not initially available.

Although planning in open-ended domains is known to be more difficult than planning based on a complete domain model, the experimental results indicate that the proposed approach scales, compared to previous planning approaches, surprisingly well to more complex domain models (see Section 9.2.2 and 9.2.3). This is mainly due to the fact that the plan-based control system automatically partitions the overall planning process into a set of simpler planning problems. It is not necessarily possible to plan more steps ahead for larger domain models, since the planning horizon is limited by the available knowledge and the lack thereof.

## 10.2. Directions of Future Research

This sections describes interesting directions for future research.

### 10.2.1. Autonomous Creation and Updating of the Domain Model

The domain models used in the context of this work are manually created by a human. However, it is desirable to enable autonomous agents to automatically create and update domain models, since this releases domain engineers from the burden of creating domain models manually and increases the autonomy of artificial agents. In fact, manually creating reasonable domain model specifications is a non-trivial task. For example, it can be difficult to answer questions like: How long does this action take in this situation? When is it better to prefer strategy A to strategy B? Instead of a priori trying to find an answer to these questions, it is desirable to enable artificial agents to deal with these questions autonomously by exploiting gathered experiences.

Some work towards the complete autonomous creation and updating of domain models has already been done. Approaches that learn parts of the domain descriptions of HTN planners (Ilghami et al., 2002, 2005, 2006; Nejati et al., 2006; Hogg et al., 2008) and various forward search planners (Yoon et al., 2008) have been proposed. Moreover, a number of approaches for learning hierarchical structures have been described in the reinforcement learning literature including McGovern and Barto (2001); Hengst (2002); Simsek and Barto (2004); Grounds and Kudenko (2007); Marthi (2006). Unfortunately, none of the aforementioned approaches have yet reached the capabilities of a human domain engineer.

### 10.2.2. Advanced Memory System

The plan-based control system presented in this work does not consider that stored information can be invalided by external events. It would be beneficial to integrate a more advanced memory system that automatically "forgets" outdated information or continually decreases the probability of acquired information.

The proposed control system does not rely on the fact that no information can be removed from the memory system. Hence, a more sophisticated memory system can be integrated without necessarily changing the proposed plan-based control system.

### 10.2.3. Ensuring Consistency

As already pointed out in Section 3.5, it is in principle possible to create a *syntactically inconsistent* (Nguyen, 2008) domain model, since the state model supports the explicit representation of negative information. On the one hand, the possible occurrence of syntactic inconsistencies is a disadvantage, since it can cause trouble for the reasoning processes of the proposed plan-based control system. On the other hand, the source of this disadvantage, namely the explicit representation of negative information, makes additional reasoning capabilities possible that can be used to detect syntactic as well as semantic inconsistencies—which can also occur in CWA-based representations—via syntactic methods.

A possible direction for future research is to develop techniques that can automatically detect inconsistencies in domain model specifications. These techniques can be used to develop tools that help domain engineers or automatic domain creation tools to detect and prevent inconsistent domain models.

### 10.2.4. Knowledge Compilation

*Knowledge compilation* (Cadoli and Donini, 1997; Darwiche and Marquis, 2002) is a general-purpose approach to reduce the computational complexity of demanding reasoning processes. The general idea of knowledge compilation is to compile a propositional theory offline into a target language so that much of the computational overhead can be pushed to this offline-phase. Online queries are answered based on the target language representation. If the queries can be answered faster based on

the target language representation than using the initial representation, then the additional computational cost of the offline-phase is amortized over an accordingly large set of online queries.

Knowledge compilation techniques have not been exploited in the context of this work. Exploring the application of knowledge compilation techniques to the domain model presented in this work constitutes a promising way to improve the performance of the proposed control system.

Appendix A

# Domain Specification

This appendix describes how domain models of the ACogControl system are specified.

The ACogControl system stores domain specifications in special *Logtalk* (Moura, 2003) objects. Logtalk is an object-oriented logic programming language that can use several Prolog implementations as a back-end compiler. Logtalk objects are transformed by the Logtalk compiler to Prolog programs. One of the supported Prolog systems can then be used to further compile or interpret these Prolog programs.

Each domain specification is an instance of a `model_datastore` object. The interface of this object is specified by the `model_datastorep` *protocol*. In Logtalk, a protocol can encapsulate predicate declarations. A protocol enables it to decouple the specification of an interface from a concrete implementation.

The `model_datastorep` protocol is shown in Listing A.1. The domain name (e.g., blocks world, or rovers) can be specified by a predicate of the form `domain(Name)`. For each literal `l` of the set of facts of a state model, there is a predicate `fact(l)` in the domain model specification. Domain specific rules are represented by predicates of the from `axiom(Head,Body)` such that `Head` is the head and `Body` is the body of the corresponding definite clause. Concept relations can be specified by a predicate

`concept(Concept,Relation,Value,Condition)`

whereby `Concept` is a literal, `Relation` is the relation (e.g., an interpretation-model-of or a subconcept-of relation), and `Condition` is a statement that is a precondition of the represented relation such that the relation only holds if an instance of this precondition is derivable. In the context of this work, the following relations are used:

- `concept(default,Relation,Value,true)`: A predicate of this form is used to define a default value (`Value`) for a concept relation (`Relation`).
- `concept(l,interpretation_model,IM,true)`: This predicate is used to ex-

plicitly define that the interpretation model of a literal `l` is `IM`.

- `concept(l,max_instances,n,c)`: This predicate equates to the predicate $i_{max_D}(\texttt{l}, \texttt{n}, \texttt{c})$ (see Section 3.9.1).
- `concept(scheme,i_max_inst_pattern,n,true)`: This predicate equates to the predicate $i_{max\_\rho}(\texttt{scheme}, \texttt{n})$ (see Section 3.9.1).
- `concept(l,is_subconcept_of,l',true)`: This predicate equates to the predicate $\texttt{l} \sqsubseteq_{def} \texttt{l'}$ (see Section 3.9.2).
- `concept(l,disjoint_with,l',true)`: This predicate equates to the predicate $\texttt{l} \sqcap_{def} \texttt{l'}$ (see Section 3.9.2).

For example, the fact that the literal `connection(Room1,Door,Room2)` has the interpretation model *cwa* can be specified by the following predicate:

```
concept(connection(_,_,_),interpretation_model,cwa,true)
```

Moreover, the following predicate can be used to define that a relation (`Relation`) holds pairwise between the concepts constituted by each literal of a set of literals (`Concepts`):

```
concept_relation(Relation,Concepts)
```

For example, the following predicate specifies that a cup, a table, a mug and a bottle are disjoint concepts:

```
concept_relation(disjoint,[cup(X),table(X),mug(X),bottle(X)])
```

The 5-tuple of a planning operator (see Definition 4.2) is specified by predicates of the following form:

```
predicate(Task,Precondition,DeleteSet,AddSet,Cost)
```

The arguments of this predicate have the same meaning as the 5-tuple defined by Definition 4.2.

HTN methods without high-level effects are specified by predicates of the form

$$\texttt{method}(\texttt{Task}, \texttt{Precondition}, \texttt{Subtasks}, \texttt{Name}, \texttt{Cost}),$$

whereas methods with high-level effects and high-level percepts are specified by predicates of the form

$$\texttt{method}(\texttt{Task}, \texttt{Precondition}, \texttt{DeleteSet}, \texttt{AddSet}, \texttt{Subtasks}, \texttt{Name}, \texttt{Percept}, \texttt{Cost}).$$

In addition to the representation of methods described in Section 4.2.3 and 4.9, the two aforementioned specification schemes additionally support the specification of a symbolic name. This symbolic name is irrelevant for the described planning algorithm (see Chapter 5). However, it can be used to more easily identify HTN methods (e.g., while monitoring or debugging the planning process).

Listing A.1: `model_datastorep` protocol that encapsulates domain model specifications.

```
:- protocol(model_datastorep,
    extends(clonablep)).

:- info([
    version is 1,
    author is 'Dominik Off',
    comment is 'A model datastore encapsulates the
        domain model specification.']).

:- public(domain/1).
:- mode(domain(-atom),one).
:- info(domain/1,[
    comment is 'Provides access to the name of the
    domain',
    argnames is ['DomainName']]).

:- public(fact/1).
:- mode(fact(?literal),zero_or_more).
:- info(fact/1,[
    comment is 'Provides access to the facts',
    argnames is ['Fact']]).

:- public(axiom/2).
:- mode(axiom(?literal,?statement),zero_or_more).
:- info(axiom/2,[
    comment is 'Provides access to the axioms of
    the domain datastore. Axioms are definite
    clauses where the head is literal and the
    body is statement.',
    argnames is ['Head','Body']]).

:- public(concept/4).
:- mode(concept(+term,+atom,-term,-statement),
    zero_or_more).
:- info(concept/4,[
    comment is 'Provides access to the conceptual
    knowledge of the datastore.
    concept(Concept,Property,Value,Condition) means that
```

```
    the property Property of the concept Concept has the
    value Value if the Condition is derivable with
    respect to the state model at hand.',
    argnames is ['Concept','Property','Value',
        'Condition']]).

:- public(concept_relation/2).
:- mode(concept_relation(+atom,+list(term)),
    zero_or_more).
:- info(concept/2,[
    comment is 'Provides access to the relations between
    concepts. For example,
    concept_relation(disjoint,[table(X),mug(X),
    bottle(X)]) represents the fact that a table,
    a mug, and a bottle a three pairwise disjunct
    concepts.',
    argnames is ['Relation','Concepts']]).

:- public(operator/5).
:- mode(operator(+term,-statement,-list(literal),
    -list(literal),-acog_expression),zero_or_more).
:- info(operator/5,[
    comment is 'Provides access to the operators of the
        domain model.',
    argnames is ['Task','Precondition','Del','Add',
        'Cost']]).

:- public(method/5).
:- mode(method(+term,-statement,-list(term),-atom,
    -acog_expression),zero_or_more).
:- info(method/5,[
    comment is 'Provides access to HTN methods of
        the domain model.',
    argnames is ['Task','Precondition','SubTasks',
        'Name','Cost']]).

:- public(method/8).
:- mode(method(+term,-statement,-list(literal),
    -list(literal),-list(term),-atom,-term,
    -acog_expression),zero_or_more).
:- info(method/8,[
    comment is 'Provides access to the HTN
        methods of the domain model.',
    argnames is ['Task','Precondition','Del','Add',
        'SubTasks','Name','Percepts','Cost']]).
```

```
85
% add and remove method for each predicate

:- public(add_fact/1).
:- public(remove_fact/1).
90 :- public(add_axiom/1).
:- public(remove_axiom/1).
:- public(add_concept/1).
:- public(remove_concept/1).
:- public(add_concept_relation/1).
95 :- public(remove_concept_relation/1).
:- public(add_operator/1).
:- public(remove_operator/1).
:- public(add_method/1).
:- public(remove_method/1).
100

% declare predicates to be dynamic so that
% they can be added and removed dynamically

105 :- dynamic(domain/1).
:- dynamic(fact/1).
:- dynamic(axiom/2).
:- dynamic(concept/4).
:- dynamic(concept_relation/2).
110 :- dynamic(operator/5).
:- dynamic(method/5).
:- dynamic(method/8).

:- end_protocol.
```

Appendix B

# Symbolic Domain Descriptions

This appendix lists the domain specifications used for the experimental evaluation.

Listing B.1 shows the domain model specification used for the real-world experiments with the service robot TASER (see Section 9.1). For clarity reasons, some variable and predicate names of the specifications shown in Chapter 7 differ slightly from the specifications shown in Listing B.1. However, these differences have no significant impact on the semantics of the domain model.

Listing B.1: Domain model specification used for the experiments with the physical robot TASER.

```
:- object(taser_datastore,
    instantiates(model_datastore)).

:- dynamic(fact/1).
:- dynamic(axiom/2).


domain(taser_restaurant).


fact(manipulator_config(joints(-90.000,-61.000,152.000,0.000,
    20.000,-90.000))).

fact(at_wp(robot,unknown)).

fact(waypoint(w1,14500,10923)).
fact(waypoint(w2,14500,12000)).
fact(waypoint(w3,14500,13500)).
fact(waypoint(w4,13500,12000)).
fact(waypoint(w5,12500,13500)).
fact(waypoint(w6,12500,12000)).
fact(waypoint(w7,11500,12000)).
fact(waypoint(w8,10500,12000)).
```

```
fact(waypoint(w9,9580,12000)).
fact(waypoint(w10,9580,10923)).
fact(waypoint(w11,13500,13000)).
fact(waypoint(w12,12500,13000)).
fact(waypoint(w13,13500,11000)).
fact(waypoint(w14,8782,13409)).
fact(waypoint(w15,8782,14536)).
fact(waypoint(w16,11900,13270)).
fact(waypoint(approach(lab_table1),10500,12970)).
fact(waypoint(center(lab_table1),10500,13270)).
fact(waypoint(center_r(lab_table1),11100,13270)).
fact(waypoint(right(lab_table1),11900,14227)).
fact(waypoint(left(lab_table1),9168,14536)).
fact(waypoint(cor1,14500,9050)).
fact(waypoint(cor2,9580,9050)).

fact(can_traverse(w1,w2)).
fact(can_traverse(w1,cor1)).
fact(can_traverse(w1,w13)).
fact(can_traverse(w2,w3)).
fact(can_traverse(w2,w4)).
fact(can_traverse(w2,w13)).
fact(can_traverse(w3,w4)).
fact(can_traverse(w3,w5)).
fact(can_traverse(w4,w6)).
fact(can_traverse(w4,w7)).
fact(can_traverse(w4,w11)).
fact(can_traverse(w4,w13)).
fact(can_traverse(w5,w6)).
fact(can_traverse(w5,w7)).
fact(can_traverse(w5,w11)).
fact(can_traverse(w5,w12)).
fact(can_traverse(w5,right(lab_table1))).
fact(can_traverse(w6,w7)).
fact(can_traverse(w6,w12)).
fact(can_traverse(w7,w8)).
fact(can_traverse(w7,w12)).
fact(can_traverse(w7,w13)).
fact(can_traverse(w8,w9)).
fact(can_traverse(w9,w10)).
fact(can_traverse(w9,w7)).
fact(can_traverse(w9,w6)).
fact(can_traverse(w9,w4)).
fact(can_traverse(w9,w2)).
fact(can_traverse(w9,w14)).
fact(can_traverse(w10,cor2)).
```

```
fact(can_traverse(w12,w11)).
fact(can_traverse(w14,w15)).
fact(can_traverse(w15,left(lab_table1))).
fact(can_traverse(approach(lab_table1),w12)).
fact(can_traverse(approach(lab_table1),w8)).
fact(can_traverse(approach(lab_table1),center(lab_table1))).
fact(can_traverse(center_r(lab_table1),center(lab_table1))).
fact(can_traverse(center_r(lab_table1),w16)).
fact(can_traverse(right(lab_table1),w16)).

fact(can_traverse_manipulator(joints(0, 0, 0, 0, 0, 0),
    modus(0.3,106),
    joints(-90.000,-61.000,152.000,
        0.000,20.000,-90.000),10)).
fact(can_traverse_manipulator(joints(0, -61, 60, 0, 20, 0),
    modus(0.5,106),
    joints(-90.000,-61.000,152.000,0.000,
        20.000,-90.000),10)).
fact(can_traverse_manipulator(joints(0, -61, 60, 0, 20, 0),
    modus(0.3,106),
    joints(119.593086, -38.642731, 84.040062, -166.434097,
        33.018036, -130.185791),10)).
fact(can_traverse_manipulator(tcp(900, 200, 930, 0, 15, 90),
    modus(0.3,106),
    joints(119.593086, -38.642731, 84.040062, -166.434097,
        33.018036, -130.185791),10)).
fact(can_traverse_manipulator(tcp(900, 200, 930, 0, 15, 90),
    modus(0.3,99),
    tcp(_X, _Y, 930, 0, 15, 90),10)).
fact(can_traverse_manipulator(tcp(X, Y, 810, 0, 15, 90),
    modus(0.1,99),
    tcp(X, Y, 930, 0, 15, 90),7)).
fact(can_traverse_manipulator(tcp(_X,_Y, 930, 0, 15, 90),
    modus(0.5,99),
    tcp(450, -300, 1100, 0, 15, 90),100)).
fact(can_traverse_manipulator(tcp(800, 0, 450, 0, 15, 90),
    modus(0.3,99),
    tcp(450, -300, 1100, 0, 15, 90),100)).
fact(can_traverse_manipulator(tcp(800, 0, 450, 0, 15, 90),
    modus(0.4,99),
    tcp(450, -300, 1100, 0, 15, 90),10)).
fact(can_traverse_manipulator(tcp(800, 0, 450, 0, 15, 90),
    modus(0.3,99),
    tcp(800, 0, 750, 0, 15, 0),10)).
fact(can_traverse_manipulator(joints(60.69095230102539,
    -13.901904106140137,85.47753143310547,
```

```
    0,58.70777893066406,-147.0784454345703),
    modus(0.7,106),
    tcp(800, 0, 750, 0, 15, 0),10)).
fact(can_traverse_manipulator(joints(60.69095230102539,
    -13.901904106140137,85.47753143310547,
    0,58.70777893066406,-147.0784454345703),modus(0.5,99),
    tcp(800, 0, 750, 0, 90, 0),10)).
fact(can_traverse_manipulator(tcp(800, 0, 750, 0, 90, 0),
    modus(0.5,106),
    tcp(800, 0, 600, 0, 90, 0),10)).

fact(room(lab)).
fact(room(corridor)).

fact(door(lab_door1)).
fact(door(lab_door2)).

fact(table(lab_table1)).
fact(table(lab_table2)).

fact(approach_pose(lab_table1,pose(center(lab_table1),
    deg(90)))).
fact(approach_pose(lab_table1,pose(center_r(lab_table1),
    deg(90)))).
fact(approach_pose(lab_table1,pose(right(lab_table1),
    deg(180)))).
fact(approach_pose(lab_table1,pose(left(lab_table1),
    deg(0)))).
fact(approach_pose(bin,pose(w4,deg(0)))).
fact(approach_pose(bin,pose(w11,deg(315)))).
fact(approach_pose(bin,pose(w13,deg(45)))).
fact(approach_pose(cross(lab,lab_door2,corridor),
    pose(cor2,deg(270)))).
fact(approach_pose(cross(lab,lab_door1,corridor),
    pose(cor1,deg(270)))).
fact(approach_pose(right(lab_table1),
        planned_final(
            pose(10800,12970,deg(90)),
            pose(10800,13270,deg(90))
        )
    )
).
fact(approach_pose(center(lab_table1),
        planned_final(
            pose(10500,12970,deg(90)),
            pose(10500,13270,deg(90))
```

```
        )
    )
).

fact(connection(lab,lab_door1,corridor)).
fact(connection(lab,lab_door2,corridor)).

fact(bottle(blue_bottle)).
fact(bottle(yellow_bottle)).
fact(bottle(big_bottle)).

fact(on(blue_bottle,lab_table1)).
fact(on(yellow_bottle,lab_table1)).
fact(on(big_bottle,lab_table1)).

fact(in_room(lab_table1,lab)).
fact(in_room(lab_table2,lab)).
fact(in_room(robot,lab)).

fact(available(ptu)).
fact(available(arm)).
fact(available(hand)).
fact(available(mobile)).
fact(available(camera)).

fact(free(hand)).


axiom(approached(W),(
    waypoint(W,_,_),approached(pose(W,_))
)).

axiom(can_traverse_manipulator(tcp(X1, Y, 810, 0, 15, 90),
modus(0.1,99),tcp(X2, Y, 930, 0, 15, 90),5),
    call((
        ground(X1),
        ground(X2)))).

axiom(manipulator_dist(tcp(X1,Y1,Z1,_,_,_),
tcp(X2,Y2,Z2,_,_,_),Dist),(
    call((
        ground(f(X1,Y1,Z1,X2,Y2,Z2)),
        XDiff is X1 - X2,
        YDiff is Y1 - Y2,
        ZDiff is Z1 - Z2,
```

```
        Dist is sqrt(XDiff * XDiff + YDiff * YDiff +
            ZDiff * ZDiff))))).

axiom(manipulator_dist(tcp(X1,Y1,Z1,_,_,_),
tcp(X2,Y2,Z2,_,_,_),10000),
    call((
        \+ ground(f(X1,Y1,Z1,X2,Y2,Z2))))).

axiom(manipulator_dist(_,_,100000),
    true).

axiom(reachable_with_arm(_X,Y),
    call((
        ground(Y)
        ->
        Y > -300
        ;
        fail))).

axiom(can_route(W1,W2,Cost),(
    call((
        ground(W1),
        ground(W2))),
    wm(WorldModel),
    call((
        once(planner::plan_atom(WorldModel,
            [route([W1],W1,W2,_R)],P)),
        P::cost(Cost))))).

axiom(nav_cost(From,Mid,To,Cost),(
    dist(From,Mid,Dist1),
    dist(Mid,To,Dist2),
    call(Cost is Dist1 + Dist2))).

axiom(nav_cost2(From,Mid,To,Cost),(
    dist(From,Mid,Dist1),
    wm(WorldModel),
    call((
        ground(Mid),
        ground(To),
        planner::plan_atom(WorldModel,
            [route([Mid],Mid,To,_)],P),
        P::cost(Dist2),
        Cost is Dist1+Dist2)))).

axiom(dist(W1,W2,Dist),(
```

```
    waypoint(W1,X1,Y1),
    waypoint(W2,X2,Y2),
    XDiff is X1 - X2,
    YDiff is Y1 - Y2,
    Dist is sqrt(XDiff * XDiff + YDiff * YDiff))).

axiom(can_traverse_manipulator(C1,M,C2,Cost),
    can_traverse_manipulator(C2,M,C1,Cost)).

axiom(direct_path(W1,W2),
    direct_path(W2,W1)).

axiom(neg direct_path(W1,W2),
    neg direct_path(W2,W1)).

axiom(can_traverse(W1,W2),
    can_traverse(W2,W1)).

axiom(can_traverse(W1,W2),(
    can_traverse(L),
    call((
        list::select(W1,L,R),
        list::select(W2,R,_))))).

axiom(approach_pose(lab_door1,pose(w1,deg(270))),(
    in_room(robot,lab))).

axiom(approach_pose(lab_door2,pose(w10,deg(270))),(
    in_room(robot,lab))).

axiom(approach_position(W,X,Y),(
    waypoint(W,X,Y))).

axiom(in_room(Object,Room),(
    on(Object,Table),in_room(Table,Room))).

axiom(connection(Room1,Door,Room2),
    connection(Room2,Door,Room1)).


concept_relation(disjoint,[in_hand(_),free(hand)]).
concept_relation(disjoint,[open(X),closed(X)]).
concept_relation(disjoint,[cup(X),table(X),mug(X),
    bottle(X)]).
concept_relation(disjoint,[mug(X),table(X),mug(X),
    bottle(X)]).
```

```
concept(default,defined,_,true).
concept(default,max_instances,unbounded,true).
concept(default,interpretation_model,owa,true).

concept(table(_),defined,_,true).
concept(cup(_),defined,_,true).
concept(closed(_),defined,_,true).
concept(in_hand(_),defined,_,true).
concept(free(hand),defined,_,true).

concept(in_room(_,_),defined,_,true).
concept(in_room(ground,var),i_max_inst_pattern,1,true).
concept(in_room(_,Room),domain,room(Room),true).

concept(on(_,_),defined,_,true).
concept(on(ground,var),i_max_inst_pattern,1,true).

concept(pos(_,_,_,_),defined,_,true).
concept(pos(ground,ground,var,var),i_max_inst_pattern,1,
    true).

concept(at_pose(_,_,_),defined,_,true).
concept(at_pose(_,_,_),max_instances,1,true).

concept(connection(_,_,_),defined,_,true).
concept(connection(_,_,_),interpretation_model,cwa,true).

concept(reachable_with_arm(_,_),defined,_,true).
concept(reachable_with_arm(_,_),interpretation_model,
    reasoning,true).

concept(manipulator_config(_),defined,_,true).
concept(manipulator_config(var),i_max_inst_pattern,1,true).

concept(approached(_),defined,_,true).
concept(approached(_),interpretation_model,cwa,true).

concept(clean(_),defined,_,true).
concept(clean(_),interpretation_model,cwa,true).

concept(seen(_),defined,_,true).
concept(seen(_),interpretation_model,cwa,true).

concept(at_wp(_,_),defined,_,true).
concept(at_wp(ground,var),i_max_inst_pattern,1,true).
```

```
346
concept(small_object(_),defined,_,true).
concept(small_object(X),is_subconcept_of,object(X),true).

concept(object(_),defined,_,true).
351 concept(object(_),is_subconcept_of,thing,true).

concept(table(_),defined,_,true).
concept(table(X),is_subconcept_of,object(X),true).

356 concept(mug(_),defined,_,true).
concept(mug(X),is_subconcept_of,small_object(X),true).
concept(mug(X),disjoint_with,table(X),true).

concept(bottle(_),defined,_,true).
361 concept(bottle(X),is_subconcept_of,small_object(X),true).

concept(room(_),defined,_,true).
concept(room(X),is_subconcept_of,location(X),true).


366
%% Activities Model

operator(set_ptu(Pan,Tilt),
    true,
371     [ptu_config(_,_)],
    [ptu_config(Pan,Tilt)],
        eval(X, call (X is 5 *6))).

operator(
376     approach_position(_X,_Y),
    true,
    [at_pose(_,_,_)],
    [],
    2).
381
operator(approach_pose_planned(pose(X,Y,Angle)),
    true,
    [at_pose(_,_,_)],
    [at_pose(X,Y,Angle)],
386     2).

operator(approach_pose_direct(pose(X,Y,Angle)),
    true,
    [at_pose(_,_,_)],
391     [at_pose(X,Y,Angle)],
```

```
    1).

operator(sense(percept(laser,
query(open(_D),_I,_C),_Response)),
    true,
    [],
    [],
    1).

operator(sense(percept(vision,
query(pos(_Object,Table,_X,_Y),_I,_C),_Response)),
    approached(Table),
    [],
    [],
    10
).
operator(sense(percept(laser,
query(rel_pos(_Dist,_Degree),_I,_C),_Response)),
    true,
    [],
    [],
    10).

operator(sense(percept(laser,
query(rel_pos(_X,_Y,_Degree),_I,_C),_Response)),
    true,
    [],
    [],
    10).

operator(sense(percept(laser,query(
free_ahead(_Dist,direct_path(_WP1,_WP2)),_I,_C),_Response)),
    true,
    [],
    [],
    10
).

operator(sense(percept(call,
query(at_pose(_X,_Y,_Degree),_I,_C),_Response)),
    true,
    [],
    [],
    1).

operator(move_forward_no_ca(_Distance),
```

```
    true,
    [at_pose(_,_,_),rel_pos(_,_),rel_pos(_,_,_)],
    [],
    5).

operator(mobile_wait_for_completed,
    true,
    [],
    [],
    5).

operator(release_brakes,
    true,
    [],
    [],
    1).

operator(apply_brakes,
    true,
    [],
    [],
    1).

operator(move_towards(_Dist,_Angle),
    true,
    [at_pose(_,_,_),rel_pos(_,_),rel_pos(_,_,_)],
    [],
    1
).

operator(rotate_rel(_Deg),
    true,
    [at_pose(_,_,_),rel_pos(_,_),rel_pos(_,_,_)],
    [],
    1).

operator(rotate_abs(_Deg),
    true,
    [at_pose(_,_,_),rel_pos(_,_),rel_pos(_,_,_)],
    [],
    1).


operator(rotate_towards(_,_),
    true,
    [at_pose(_,_,_),rel_pos(_,_),rel_pos(_,_,_)],
```

```
    [],
    5).

operator(start_trajectory_generator,
    available(arm),
    [],
    [],
    1).

operator(stop_trajectory_generator,
    available(arm),
    [],
    [],
    1).

operator(open_fingers,
    true,
    [],
    [],
    3).

operator(close_fingers,
    true,
    [],
    [],
    3).

operator(set_arm_tcp(X,Y,Z,Angle1,Angle2,Angle3,
_Speed,_Mode),
    true,
    [manipulator_config(_)],
    [manipulator_config(tcp(X,Y,Z,Angle1,Angle2,Angle3))],
    10).

operator(set_arm_joints(J1,J2,J3,J4,J5,J6,_Speed,_Mode),
    true,
    [manipulator_config(_)],
    [manipulator_config(joints(J1,J2,J3,J4,J5,J6))],
    5
).

operator(do_route(From,To),
    true,
    [],
    [],
    eval(Dist,(dist(From,To,Dist1), call(Dist is Dist1)))).
```

```
531
% HTN methods

method(move_manipulator(GoalConfig),
    manipulator_config(CurrentConfig),
    [
        start_trajectory_generator,
        move_manipulator([CurrentConfig],GoalConfig),
        stop_trajectory_generator],
    no,
    0).

method(move_manipulator(_Visited,GoalConfig),
    (
        manipulator_config(GoalConfig)),
    [],
    default_name,
    0).

method(move_manipulator(_Visited,GoalConfig),
    (
        manipulator_config(CurrentConfig),
        GoalConfig \= CurrentConfig,
        can_traverse_manipulator(CurrentConfig,
            Modus,GoalConfig,C)),
    [direct_move_manipulator(Modus,GoalConfig)],
    default_name,
    C).

method(
    move_manipulator([LastConf|Visited],GoalConfig),
    (
        manipulator_config(CurrentConfig),
        GoalConfig \= CurrentConfig,
        can_traverse_manipulator(CurrentConfig,Modus,
            SomeConfig,Cost),
        call((
            % ensure that no identical config is a member
            % of [LastConf|Visited]
            \+ extlist::id_member(SomeConfig,
                [LastConf|Visited]),
            \+ list::member(CurrentConfig,Visited)))
    ),
    [
```

```
576         direct_move_manipulator(Modus,SomeConfig),
            move_manipulator([SomeConfig|[LastConf|Visited]],
                GoalConfig)],
    default_name,
    Cost).
581
method(direct_move_manipulator(modus(Speed,Mode),
joints(J1,J2,J3,J4,J5,J6)),
    true,
    [set_arm_joints(J1,J2,J3,J4,J5,J6,Speed,Mode)],
586 default_name,
    10).

method(direct_move_manipulator(modus(Speed,Mode),
tcp(P1,P2,P3,Angle1,Angle2,Angle3)),
591 true,
    [
        set_arm_tcp(P1,P2,P3,Angle1,
            Angle2,Angle3,Speed,Mode)],
    default_name,
596 10).

method(move_to(Room),
    in_room(robot,Room),
    [],
601 in_goalroom,
    0).

method(move_to(Room),
    (
606     in_room(robot,Other),
        connection(Other,Door,Room),
        open(Door),
        at_wp(robot,From),
        approach_pose(Door,pose(WP,_)),
611     can_route(WP,From,RouteCost)
    ),
    [cross(Other,Door,Room)],
    direct_connection,
    RouteCost).
616
method(move_to(Room),
    (in_room(robot,Room1), corridor \= Room,
        corridor\=Room1),
    [move_to(corridor),move_to(Room)],
621 else_to_corridor,
```

```
    1000000000).

method(pick_up(Object),
    (
        in_hand(Object)
    ),
    [],
    nothing_to_do,
    0).

method(pick_up(Object),
    (
        free(hand),
        small_object(Object),
        table(Table),
        on(Object,Table),
        at_wp(robot,From),
        From \= unknown,
        approach_pose(Table,pose(Waypoint,Angle)),
        pos(Object,pose(Waypoint,Angle),X,Y),
        reachable_with_arm(X,Y),
        can_route(Waypoint,From,RouteCost)),
    [
        navigate(pose(Waypoint,Angle)),
        grab(Object,pose(Waypoint,Angle))],
    all_known,
    RouteCost).

method(graspFromTable(X,Y,_Table),
    (
        Y < 200,
        XApproach is X -100,
        XGrasp is X - 10,
        Y2 is Y - 50),
    [
        close_fingers,
        move_manipulator(tcp(XApproach, Y2, 930, 0, 15, 90)),
        open_fingers,
        move_manipulator(tcp(XGrasp, Y2, 810, 0, 15, 90)),
        close_fingers,
        move_manipulator(tcp(450, -300, 1100, 0, 15, 90))],
    default_name,
    100).

method(approach_bucket(Dist,_Angle),
    (
```

```
        Dist < 600,
        DriveDist is (Dist - 640)),
    [
        move_forward_no_ca(DriveDist),
        mobile_wait_for_completed,
        bin],
    to_close,
    100).

method(approach_bucket(Dist,Angle),
    (
        Dist > 680,
        Dist < 1680,
        DriveDist is Dist - 640),
    [
        move_towards(DriveDist,Angle),
        mobile_wait_for_completed,
        bin],
    to_far,
    100).

method(approach_bucket(Dist,Angle),
    (
        Dist > 600,
        Dist < 680),
    [
        rotate_rel(Angle),
        mobile_wait_for_completed,
        apply_brakes,
        move_manipulator(tcp(800, 0, 450, 0, 15, 90)),
        open_fingers],
    dist_ok,
    100).

method(approach_grasp_bucket,
    (
      rel_pos(Dist,Angle)),
    [approach_grasp_bucket(Dist,Angle)],
    default_name,
    100).

method(approach_grasp_bucket(Dist,_Angle),
    (
        Dist < 600,
        DriveDist is (Dist - 640)),
    [
```

```
        move_forward_no_ca(DriveDist),
        mobile_wait_for_completed,
        approach_grasp_bucket],
    to_near,
    100).

method(approach_grasp_bucket(Dist,Angle),
    (
        Dist > 680,
        Dist < 1680,
        DriveDist is Dist - 680),
    [
        move_towards(DriveDist,Angle),
        mobile_wait_for_completed,
        approach_grasp_bucket],
    to_far,
    100).

method(grasp_bucket,
    (
        rel_pos(X,Y,Angle)),
    [grasp_bucket(X,Y,Angle)],
    default_name,
    100).

method(route(To,Route),
    at_wp(robot,From),
    [route([From],From,To,Route)],
    nothing,
    0).

method(route(Visited,From,To,Route),
    true,
    [route1(Visited,From,To,Route)],
    r1,
    1).

method(route(Visited,From,To,Route),
    true,
    [route2(Visited,From,To,Route)],
    r2,
    2).

method(route(Visited,From,To,Route),
    true,
    [route3(Visited,From,To,Route)],
```

```
    r3,
    3).

method(route1(_,To,To,[]),
    true,
    [],
    nothing,
    0).

method(route2(Visited,From,To,[To]),
    (
        can_traverse(From,To),
        To notin Visited,
        possibly direct_path(From,To)),
    [do_route(From,To)],
    do2,
    eval(Cost,dist(From,To,Cost))).

method(route3(Visited,From,To,[Mid|Route]),
    (
         can_traverse(From,Mid),
         Mid notin Visited,
         possibly direct_path(From,Mid)),
    [
        do_route(From,To),
        route([Mid|Visited],Mid,To,Route)],
    do3,
    eval(Cost,
        (
            dist(Mid,To,Dist),
            Cost is Dist + 10000))).

method(navigate(_,_,To),
    at_wp(robot,To),
    [],
    nothing,
    0).

method(navigate(_,From,To),
    (
        at_wp(robot,From), can_traverse(From,To),
        direct_path(From,To)),
    [
        do_navigate(From,To)],
    do2,
    1).
```

```
806
method(navigate(Visited,From,To),
    (
        at_wp(robot,From),
        can_traverse(From,Mid),
811     Mid \= To,
        Mid notin Visited,
        direct_path(From,Mid)),
    [
        do_navigate(From,Mid),
816     navigate([Mid|Visited],Mid,To)],
    do3,
    eval(Cost,nav_cost(From,Mid,To,Cost))).

method(navigate_via(Visited,From,Mid,To),
821 (
        can_route(To,Mid,_CostMid)
    ),
    [do_navigate(From,Mid), navigate([Mid|Visited],Mid,To)],
    default_name,
826 5).

method(determine(percept(vision,
query(pos(Object,Table,X,Y),I,C),Response)),
    true,
831 [
        mobile_wait_for_completed,
        navigate(Table),
        mobile_wait_for_completed,
        move_manipulator(joints(0, -61, 60, 0, 20, 0)),
836     sense(percept(vision,
            query(pos(Object,Table,X,Y),I,C),
        Response))],
    default_name,
    500).
841
method(determine(percept(laser,
query(rel_pos(Dist,Angle),I,C),Response)),
    true,
    [
846     mobile_wait_for_completed,
        sense(percept(laser,
            query(rel_pos(Dist,Angle),I,C),Response))],
    default_name,
    50).
851
```

```
method(determine(percept(laser,
query(rel_pos(X,Y,Angle),_I,_C),Response)),
    true,
    [
        mobile_wait_for_completed,
        sense(percept(laser,
            query(rel_pos(X,Y,Angle),_I,_C),Response))],
    default_name,
    50).

method(determine(percept(laser,
query(open(D),I,C),Response)),
    (
        approach_pose(D,Goal)),
    [
        navigate(Goal),
        mobile_wait_for_completed,
        sense(percept(laser,query(open(D),I,C),Response))],
    default_name,
    50).

method(determine(percept(call,
query(at_pose(X,Y,Degree),I,C),Response)),
    true,
    [
        sense(percept(call,
            query(at_pose(X,Y,Degree),I,C),Response))],
    default_name,
    50).

method(determine(percept(laser,
query(direct_path(WP1,WP2),I,C),response(L,[]))),
    (
        waypoint(WP1,X1,Y1),
        waypoint(WP2,X2,Y2)),
    [],
    [],
    [
        approach(WP1),
        determine(percept(laser,query(reachable(X1,Y1,X2,Y2,
            direct_path(WP1,WP2)),I,C),Response))],
    default_name,
    eval(percept(laser,query(direct_path(WP1,WP2),I,C),
        response(L,[])),call((
        Response::literal(reachable(X1,Y1,X2,Y2,
        direct_path(WP1,WP2)))
```

```
        ->
        L = direct_path(WP1,WP2)
        ;
        L = neg direct_path(WP1,WP2)))),
    100).

method(determine(percept(laser,query(reachable(X1,Y1,X2,Y2,
direct_path(WP1,WP2)),I,C),response(L,[]))),
    (
        XDiff is X1 - X2,
        YDiff is Y1 - Y2,
        Distance is sqrt(XDiff * XDiff + YDiff * YDiff)),
    [],
    [],
    [
        mobile_wait_for_completed,
        rotate_towards(X2,Y2),
        mobile_wait_for_completed,
        sense(percept(laser,query(free_ahead(Distance1,
            direct_path(WP1,WP2)),[],C),Response))],
    default_name,
    eval(percept(laser,query(reachable(X1,Y1,X2,Y2,
    direct_path(WP1,WP2)),I,C),response(L,[])),call((
        (Response::literal(free_ahead(Distance1,
        direct_path(WP1,WP2))), Distance < Distance1)
        ->
        L = reachable(X1,Y1,X2,Y2,direct_path(WP1,WP2))
        ;
        L = neg reachable(X1,Y1,X2,Y2,direct_path(WP1,WP2))
    ))),
    100).

method(approach_grasp_bucket(Dist,_Angle),
    (
      Dist > 600,
      Dist < 680),
    [approached(_)],
    [approached(grasp_bin)],
    [],
    default_name,
    no_percepts,
    100).

method(cross(From,Door,To),
    (
        in_room(robot,From),
```

```
        approach_pose(cross(From,Door,To),Pose)),
    [in_room(robot,From)],
    [in_room(robot,To)],
    [navigate(Pose)],
    default_name,
    no_percepts,
    5).

method(grab(Object,Table),
    (
        free(hand),
        pos(Object,Table,X,Y)),
    [free(hand)],
    [in_hand(Object)],
    [graspFromTable(X,Y,Table)],
    default_name,
    no_percepts,
    5).

method(grab(Object,_Table),
    in_hand(Object),
    [],
    [],
    [],
    nothing_to_do,
    no_percepts,
    0).

method(approach(Entity),
    approached(Entity),
    [],
    [],
    [],
    default_name,
    no_percepts,
    0).

method(approach(Entity),
    (
        \+ approached(Entity),
        approach_position(Entity,X,Y)),
    [approached(_)],
    [approached(Entity)],
    [
        mobile_wait_for_completed,
        approach_position(X,Y)],
```

```
    default_name,
    no_percepts,
    50).

method(approach(Entity),
(
    neg approached(Entity),
    approach_pose(Entity,
        planned_final(
            pose(X,Y,Angle),
            pose(X2,Y2,Angle2)))),
[approached(_)],
[approached(Entity)],
[
    approach_pose_planned(pose(X,Y,Angle)),
    approach_pose_direct(pose(X2,Y2,Angle2))],
default_name,
no_percepts,
50).

method(approach(Entity),
    (
        neg approached(Entity),
        approach_pose(Entity,pose(X,Y,Angle))),
    [approached(_)],
    [approached(Entity)],
    [approach_pose_planned(pose(X,Y,Angle))],
    default_name,
    no_percepts,
    35).

method(
    bin,
    rel_pos(Dist,Angle),
    [in_hand(X)],
    [free(hand),
    in_bin(X)],
    [
        release_brakes,
        approach_bucket(Dist,Angle)],
    default_name,
    no_percepts,
    50).

method(pick_up_bucket,
    in_hand(garbage_can),
```

```
    [],
    [],
    [],
    default_name,
    no_percepts,
    0).

method(pick_up_bucket,
    (
        free(hand),
        rel_pos(Dist,Angle),
        neg approached(grasp_bin)),
    [],
    [],
    [
        approach_grasp_bucket(Dist,Angle),
        grasp_bucket],
    default_name,
    no_percepts,
    50).

method(
    pick_up_bucket,
    (
        free(hand),
        approached(grasp_bin)),
    [],
    [],
    [grasp_bucket],
    default_name,
    no_percepts,
    50).

method(put_down_garbage_can,
    in_hand(_),
    [in_hand(_)],
    [free(hand)],
    [
        mobile_wait_for_completed,
        start_trajectory_generator,
        set_arm_tcp(600, -250, 950, 0,90,-90, 0.5, 99),
        set_arm_tcp(600, 150, 420, 0,90,-90, 0.3, 99),
        set_arm_tcp(600, -150, 370, 0,90,-90, 0.2, 99),
        open_fingers,
        set_arm_joints(-90.000,-61.000,152.000,0.000,
            20.000,-90.000,0.3,106),
```

```
        stop_trajectory_generator],
    default_name,
    no_percepts,
    100).

method(put_down_garbage_can,
    free(hand),
    [],
    [],
    [],
    default_name,
    no_percepts,
    100).

method(clean_obj(Obj),
    (
        on(Obj,_Table)),
    [],
    [],
    [
        pick_up(Obj),
        navigate_to_entity(bin),
        bin],
    on_table,
    no_percepts,
    5).

method(clean_obj(Obj),
    (
        in_hand(Obj)),
    [],
    [],
    [
        navigate_to_entity(bin),
        bin],
    in_hand,
    no_percepts,
    2).

method(clean_obj(Obj),
    in_bin(Obj),
    [],
    [],
    [],
    nothing_to_do,
    no_percepts,
```

```
    0).

method(clean(Table),
    (
        neg clean(Table),
        free(hand),
        on(Obj,Table)),
    [],
    [clean(Table)],
    [
        pick_up(Obj),
        navigate_to_entity(bin),
        bin],
    hand_free,
    no_percepts,
    5).

method(clean(Table),
    (
        neg clean(Table),
        neg free(hand)),
    [],
    [clean(Table)],
    [
        navigate_to_entity(bin),
        bin],
    in_hand,
    no_percepts,
    2).

method(clean(Table),
    clean(Table),
    [],
    [],
    [],
    nothing_to_do,
    no_percepts,
    0).

method(navigate_to_entity(E),
    (Except the adaption of
    the set of facts all
        at_wp(robot,From),
        From \= unknown ,
        approach_pose(E,pose(Waypoint,Angle)),
        can_route(Waypoint,From,RouteCost)),
```

```
    [],
    [],
    [navigate(pose(Waypoint,Angle))],
    default_name,
    no_percepts,
    RouteCost).

method(grasp_bucket(X,Y,_Angle),
    true,
    [free(hand)],
    [in_hand(garbage_can)],
    [

       open_fingers,
       move_manipulator(tcp(800, 0, 600, 0, 90, 0)),
       start_trajectory_generator,
       set_arm_tcp(X,Y,320, 0, 90, 0, 0.1, 106),
       close_fingers,
       set_arm_tcp(600, -250, 950,0, 90, -90, 0.2, 99),
       set_arm_tcp(250, -210, 1100,0, 90, -90, 0.5, 99),
       stop_trajectory_generator],
    default_name,
    no_percepts,
    100).

method(clearance(_),
    true,
    [],
    [],
    [
        clean_obj(blue_bottle),
        clean_obj(yellow_bottle),
        clean_obj(big_bottle),
        clean_obj(big_bottle2),
        clean_obj(big_bottle3),
        pick_up_bucket,
        move_to(corridor),
        put_down_garbage_can],
    default_name,
    no_percepts,
    20).


method(
    do_navigate(From,To),
    true,
```

```
    [at_wp(robot,From)],
    [at_wp(robot,To)],
    [approach(To)],
    default_name,
    no_percepts,
    1).

method(
    go_on_roadmap(From),
    (
        at_wp(robot,From),
        From \= unknown),
    [],
    [],
    [],
    default_name,
    no_percepts,
    0).

method(
    go_on_roadmap(W),
    (
        at_wp(robot,unknown),
        at_pose(X,Y,_Angle),
        waypoint(W,WX,WY)),
    [at_wp(robot,unknown)],
    [at_wp(robot,W)],
    [approach_pose_planned(pose(WX,WY,deg(0)))],
    default_name,
    no_percepts,
    eval(Cost,(
        call((
            (ground(X),ground(Y)) ->
                (
                    XDiff is X - WX,
                    YDiff is Y - WY,
                    Cost is sqrt(XDiff * XDiff
                        + YDiff * YDiff)
                )
                ;
                Cost = 100000000))))).

method(ensure_drive_pose,
    (
        free(hand),
        neg manipulator_config(joints(-90.000,-61.000,
```

```
            152.000,0.000,20.000,-90.000))),
    [],
    [drive_pose],
    [move_manipulator(joints(-90.000,-61.000,
        152.000,0.000,20.000,-90.000))],
    default_name,
    no_percepts,
    10000).

method(ensure_drive_pose,
    (
        in_hand(_)
        ;
        manipulator_config(joints(-90.000,-61.000,
            152.000,0.000,20.000,-90.000))
    ),
    [],
    [],
    [],
    nothing_to_do,
    no_percepts,
    0).

method(navigate(pose(To,deg(_Angle))),
    at_wp(robot,To),
    [],
    [],
    [],
    nothing,
    no_percepts,
    0).

method(navigate(To),
    (waypoint(To,_,_)),
    [approached(_)],
    [approached(To)],
    [
        ensure_drive_pose,
        go_on_roadmap(From),
        navigate([From],From,To)],
    do_nav,
    no_percepts,
    10000).

method(navigate(pose(To,deg(Angle))),
    (
```

```
    neg at_wp(robot,To),
    waypoint(To,_,_)),
    [approached(_)],
    [approached(pose(To,deg(Angle)))],
    [
        ensure_drive_pose,
        go_on_roadmap(From),
        navigate([From],From,To),
        rotate_abs(Angle),
        mobile_wait_for_completed],
    do_nav2,
    no_percepts,
    10000).

:- end_object.
```

For the simulated-based experiments described in Section 9.2.2, the domain specification shown in Listing B.1 is successively extended by additional facts. Except for adaptions of the set of facts, all other parts of the domain model specification are identical for all experiments with the service robotic domain.

Listing B.2 shows a domain model specification for a small instance (i.e., with a small number of facts) of the blocks world domain. For clarity reasons, this instance is significantly smaller than the instances that were used for the experiments. Except for adaptions of the set of facts, all other parts of the domain model specification are identical for all experiments with this blocks world domain.

Listing B.2: Domain model specification of a small instance of the blocks world domain.

```
:- object(blocks_datastore,
    instantiates(model_datastore)).

:- dynamic(fact/1).
:- dynamic(axiom/2).

domain(blocks).

fact(on(b15,b20)).
fact(on(b24,b9)).
fact(on(b20,b21)).
fact(on(b21,b1)).
fact(on(b9,b23)).
fact(on(b6,b22)).
fact(on(b22,b8)).
fact(on(b12,b25)).
fact(on(b23,b2)).
fact(on(b2,b16)).
```

```
fact(on(b8,b11)).
fact(on(b25,b10)).
fact(on(b1,b4)).
fact(on(b10,b17)).
fact(on(b17,b7)).
fact(on(b11,b14)).
fact(on(b7,b18)).
fact(on(b4,b5)).
fact(on_table(b5)).
fact(on_table(b14)).
fact(on(b18,b13)).
fact(on(b16,b19)).
fact(on(b13,b3)).
fact(on_table(b19)).
fact(on_table(b3)).
fact(clear(b6)).
fact(clear(b12)).
fact(clear(b15)).
fact(clear(b24)).

axiom(
    neg clear(Y),
    on(_,Y)).

axiom(
    neg on(_,Y),
    clear(Y)).

axiom(
    neg on(X,Y),
    (
        on(X,Z),
        Z \= Y
    )
).

axiom(neg on(X,Y),
    (
        on(Z,Y),
        Z \= X)).


concept(default/0,defined,_,true).
concept(default,max_instances,unbounded,true).
concept(default,interpretation_model,owa,true).

```

```
concept(on/2,defined,_,true).
concept(on(Object,_),max_instances,1,ground(Object)).
concept(on(_,Object),max_instances,1,ground(Object)).


operator(pickup(A),
    (
        clear(A),
        on_table(A)),
    [
        clear(A),
        on_table(A)],
    [holding(A)],
    1).

operator(putdown(B),
    (holding(B)),
    [holding(B)],
    [on_table(B),clear(B)],
    1).

operator(
    stack(C,D),
    (holding(C),clear(D)),
    [holding(C),clear(D)],
    [on(C,D),clear(C)],
    1).

operator(
    unstack(E,F),
    (clear(E),on(E,F)),
    [clear(E),on(E,F)],
    [holding(E),clear(F)],
    1).

operator(
    sense(percept(percept,_,_)),
    true,
    [],
    [],
    1).


method(
    put_on(X,Y),
    (on(X,Y)),
```

```
    [],
    nothing_to_do,
    0).

method(
    put_on(X,Y),
    true,
    [make_clear(X),make_clear(Y),move(X,Y)],
    make_clear,
    0.1).

method(
    make_clear(Block),
    clear(Block),
    [],
    nothing_left,
    0).

method(
    make_clear(Block),
     on(Block2,Block),
    [make_clear(Block2),unstack(Block2,Block),
        putdown(Block2)],
    putdown_indirect_on,
    0.1).

method(
    move(X,Y),
    (clear(Y),clear(X),on(X,Z)),
    [unstack(X,Z),stack(X,Y)],
    on_table,
    1).

method(
    move(X,Y),
    (clear(Y),clear(X),on_table(X)),
    [pickup(X),stack(X,Y)],
    on_table2,
    1).

method(determine(percept(percept,query(Q,I,C),Response)),
    true,
    [sense(percept(percept,query(Q,I,C),Response))],
    dc1,
    5000).
```

```
:- end_object.
```

Listing B.3 shows a domain model specification for a small instance (i.e., with a small number of facts) of the depots domain. For clarity reasons, this instance is significantly smaller than the instances that were used for the experiments. Except for adaptions of the set of facts, all other parts of the domain model specification are identical for all experiments with this depots domain.

Listing B.3: Domain model specification of a small instance of the depots domain.

```
:- object(depots_datastore,
    instantiates(model_datastore)).

:- dynamic(fact/1).
:- dynamic(axiom/2).

domain(depots).

fact(depot(depot0)).
fact(distributor(distributor0)).
fact(distributor(distributor1)).
fact(truck(truck0)).
fact(truck(truck1)).
fact(pallet(pallet0)).
fact(pallet(pallet1)).
fact(pallet(pallet2)).
fact(crate(crate0)).
fact(crate(crate1)).
fact(crate(crate2)).
fact(crate(crate3)).
fact(crate(crate4)).
fact(crate(crate5)).
fact(hoist(hoist0)).
fact(hoist(hoist1)).
fact(hoist(hoist2)).
fact(at(pallet0,depot0)).
fact(clear(crate1)).
fact(at(pallet1,distributor0)).
fact(clear(crate4)).
fact(at(pallet2,distributor1)).
fact(clear(crate5)).
fact(at(truck0,depot0)).
fact(at(truck1,distributor0)).
fact(at(hoist0,depot0)).
fact(available(hoist0)).
fact(at(hoist1,distributor0)).
fact(available(hoist1)).
```

```
fact(at(hoist2,distributor1)).
fact(available(hoist2)).
fact(at(crate0,distributor0)).
fact(on(crate0,pallet1)).
fact(at(crate1,depot0)).
fact(on(crate1,pallet0)).
fact(at(crate2,distributor1)).
fact(on(crate2,pallet2)).
fact(at(crate3,distributor0)).
fact(on(crate3,crate0)).
fact(at(crate4,distributor0)).
fact(on(crate4,crate3)).
fact(at(crate5,distributor1)).
fact(on(crate5,crate2)).


axiom(neg at(Crate,Location),
    (
        at(Crate,Location1),
        Location1\=Location)).

axiom(neg on(X,Y),
    (
        on(X,Z),
        Z \= Y)).

axiom(neg on(X,Y),
    (
        on(Z,Y),
        Z \= X)).

axiom(neg lifting(_,Y),
    (
        at(Y,_))).

axiom(neg clear(Y),
    (
        on(Y,_))).

concept_relation(disjoint,
    [hoist(X),truck(X),crate(X),pallet(X)]).
concept_relation(disjoint,[available(X),lifting(X,_)]).
concept_relation(disjoint,[on(_,X),clear(X)]).


concept(default,defined,_,true).
```

```
concept(default,max_instances,unbounded,true).
concept(default,interpretation_model,owa,true).

concept(at(_,_),defined,_,true).
concept(at(ground,var),i_max_inst_pattern,1,true).

concept(clear(_),defined,_,true).
concept(clear(X),is_subconcept_of,crate(X),true).

concept(lifting(_,_),defined,_,true).
concept(lifting(X,_),max_instances,0,
    (call(ground(X)),available(X))).

concept(on(_,_),defined,_,true).
concept(on(_,X),max_instances,0,(call(ground(X)),clear(X))).

concept(object(_),defined,_,true).
concept(object(_),is_subconcept_of,thing,true).

concept(locatable(_),defined,_,true).
concept(locatable(X),is_subconcept_of,object(X),true).

concept(place(_),defined,_,true).
concept(place(X),is_subconcept_of,object(X),true).

concept(depot(_),defined,_,true).
concept(depot(X),is_subconcept_of,place(X),true).

concept(distributor(_),defined,_,true).
concept(distributor(X),is_subconcept_of,place(X),true).

concept(truck(_),defined,_,true).
concept(truck(X),is_subconcept_of,locatable(X),true).

concept(hoist(_),defined,_,true).
concept(hoist(X),is_subconcept_of,locatable(X),true).

concept(available(_),defined,_,true).
concept(available(X),is_subconcept_of,hoist(X),true).

concept(surface(_),defined,_,true).
concept(surface(X),is_subconcept_of,locatable(X),true).

concept(pallet(_),defined,_,true).
concept(pallet(X),is_subconcept_of,surface(X),true).
```

```
concept(crate(_),defined,_,true).
concept(crate(X),is_subconcept_of,surface(X),true).
concept(crate(X),disjoint_with,hoist(X),true).


operator(drive(Truck,Start,Dest),
    once at(Truck,Start),
    [at(Truck,Start)],
    [at(Truck,Dest)],
    1).

operator(lift(Hoist,Crate,Surface,Place),
    once (at(Hoist,Place),available(Hoist),at(Crate,Place),
        on(Crate,Surface),clear(Crate)),
    [at(Crate,Place),clear(Crate),available(Hoist),
        on(Crate,Surface)],
    [lifting(Hoist,Crate),clear(Surface)],
    1).

operator(drop(Hoist,Crate,Surface,Place),
    once (
        at(Hoist,Place),
        at(Surface,Place),
        clear(Surface),
        lifting(Hoist,Crate)),
    [lifting(Hoist,Crate),clear(Surface)],
    [
        available(Hoist),
        at(Crate,Place),
        clear(Crate),
        on(Crate,Surface)],
    1).

operator(load(Hoist,Crate,Truck,Place),
    once (
        lifting(Hoist,Crate),
        truck(Truck),
        at(Hoist,Place),
        at(Truck,Place)),
    [lifting(Hoist,Crate)],
    [in(Crate,Truck),available(Hoist)],
    1).

operator(unload(Hoist,Crate,Truck,Place),
    once (
```

```
        at(Hoist,Place),
        at(Truck,Place),
        available(Hoist),
        in(Crate,Truck)),
    [in(Crate,Truck),available(Hoist)],
    [lifting(Hoist,Crate)],
    1).

operator(sense(percept(percept,query(_X,_I,_C),_Response)),
    true,
    [],
    [],
    1).

operator(sense(query(at(_A,_B),_,_),perception),
    true,
    [],
    [],
    10).


method(deliver(Crate,Place),
    at(Crate,Place),
    [],
    nothing_to_do,
    0).

method(deliver(Crate,_Place),
    (lifting(Hoist,Crate)),
    [drop(Hoist,Crate,_Surface2,_)],
    default_name,
    10).

method(deliver(Crate,Place),
    in(Crate,Truck),
    [
        drive(Truck,_,Place),
        make_hoist_available(Place),
        unload(_H,Crate,Truck,Place),
        drop(_Hoist2,Crate,_Surface2,Place)],
    default_name,
    10).

method(
    deliver(Crate,Place),
    (
```

```
        at(Crate,Place2),
        Place2\=Place),
    [
        make_clear(Crate,Place2),
        move(Crate,Place)],
    default_name,
    100).

method(move(Crate,Place),
    at(Crate,Place),
    [],
    default_name,
    0).

method(move(Crate,Place),
    (
        neg at(Crate,Place),
        at(Crate,Place2)),
    [
        make_truck_available(Place2),
        lift(Hoist,Crate,_Surface,Place2),
        load(Hoist,Crate,Truck,Place2),
        drive(Truck,Place2,Place),
        make_hoist_available(Place),
        unload(_Hoist2,Crate,Truck,Place),
        drop(_Hoist2,Crate,_Surface2,Place)],
    default_name,
    10).

method(make_truck_available(Place),
    once (
        truck(Truck),
        at(Truck,Place)),
    [],
    nothing_to_do,
    0).

method(make_truck_available(Place),
    once (
        truck(Truck),
        neg at(Truck,Place)),
    [drive(Truck,_Start,Place)],
    drive,
    0.1).

method(make_hoist_available(Place),
```

```
268     once (
            available(H),
            at(H,Place)),
        [],
        nothing_to_do,
273     0).

    method(make_hoist_available(Place),
        once (
            hoist(H),
278         at(H,Place),
            lifting(H,C)),
        [drop(H,C,_Surface,Place)],
        drop,
        10).
283
    method(make_clear(Surface,_),
        clear(Surface),
        [],
        nothing_left,
288     0).

    method(make_clear(Surface,Place),
        once (on(Surface2,Surface),place(Place2),Place2\=Place),
        [make_clear(Surface2,Place),move(Surface2,Place2)],
293     make_clear_top,
        0.1).

    method(determine(percept(percept,query(X,I,C),Response)),
        true,
298     [sense(percept(percept,query(X,I,C),Response))],
        default_name,
        5000).

    :- end_object.
```

Listing B.4 shows a domain model specification for a small instance (i.e., with a small number of facts) of the rovers domain. For clarity reasons, this instance is significantly smaller than the instances that were used for the experiments. Except for adaptions of the set of facts, all other parts of the domain model specification are identical for all experiments with this rovers domain.

The external method

```
acog_reasoner::not_tried_without_gain_as(AS)
```

(used in the domain specification) determines whether there is no unaware (see

Definition 4.11) knowledge source that renders the knowledge acquisition scheme (AS) useless. Explicitly calling this reasoning procedure can be necessary if the substitutions that are applied to a precondition during the instantiation process also instantiate the knowledge acquisition task and in this way possibly make corresponding knowledge sources unaware.

Listing B.4: Domain model specification of a small instance for the rovers domain.

```
:- object(rovers_datastore,
    instantiates(model_datastore)).

    :- dynamic(fact/1).
    :- dynamic(axiom/2).

domain(rovers).

fact(approach_percept_query(at_soil_sample(W),W)).
fact(approach_percept_query(at_rock_sample(W),W)).

fact(visible(waypoint1,waypoint0)).
fact(visible(waypoint0,waypoint1)).
fact(visible(waypoint2,waypoint0)).
fact(visible(waypoint0,waypoint2)).
fact(visible(waypoint2,waypoint1)).
fact(visible(waypoint1,waypoint2)).
fact(visible(waypoint3,waypoint0)).
fact(visible(waypoint0,waypoint3)).
fact(visible(waypoint3,waypoint1)).
fact(visible(waypoint1,waypoint3)).
fact(visible(waypoint3,waypoint2)).
fact(visible(waypoint2,waypoint3)).
fact(at_soil_sample(waypoint0)).
fact(at_rock_sample(waypoint1)).
fact(at_soil_sample(waypoint2)).
fact(at_rock_sample(waypoint2)).
fact(at_soil_sample(waypoint3)).
fact(at_rock_sample(waypoint3)).
fact(at_lander(general,waypoint0)).
fact(channel_free(general)).
fact(at(rover0,waypoint3)).
fact(available(rover0)).
fact(store_of(rover0store,rover0)).
fact(empty(rover0store)).
fact(equipped_for_soil_analysis(rover0)).
fact(equipped_for_rock_analysis(rover0)).
fact(equipped_for_imaging(rover0)).
```

```
fact(can_traverse(rover0,waypoint3,waypoint0)).
fact(can_traverse(rover0,waypoint0,waypoint3)).
fact(can_traverse(rover0,waypoint3,waypoint1)).
fact(can_traverse(rover0,waypoint1,waypoint3)).
fact(can_traverse(rover0,waypoint1,waypoint2)).
fact(can_traverse(rover0,waypoint2,waypoint1)).
fact(on_board(camera0,rover0)).
fact(calibration_target(camera0,objective1)).
fact(supports(camera0,colour)).
fact(supports(camera0,high_res)).
fact(visible_from(objective0,waypoint0)).
fact(visible_from(objective0,waypoint1)).
fact(visible_from(objective0,waypoint2)).
fact(visible_from(objective0,waypoint3)).
fact(visible_from(objective1,waypoint0)).
fact(visible_from(objective1,waypoint1)).
fact(visible_from(objective1,waypoint2)).
fact(visible_from(objective1,waypoint3)).


% can_traverse is symmetric
axiom(can_traverse(R,W1,W2),
    can_traverse(R,W2,W1)).

% visible is symmetric
axiom(visible(W1,W2),
    visible(W2,W1)).


concept(default/0,defined,_,true).
concept(default,max_instances,unbounded,true).
concept(default,interpretation_model,owa,true).

concept(approach_percept_query/2,defined,_,true).
concept(approach_percept_query(_,_),interpretation_model,
    cwa,true).

concept(at_lander/2,defined,_,true).
concept(at_lander(_,_),max_instances,1,true).

concept(at/2,defined,_,true).
concept(at(ground,var),i_max_inst_pattern,1,true).

concept(visited/1,defined,_,true).
concept(visited(X),interpretation_model,cwa,call(ground(X))).
```

```
concept(seen/1,defined,_,true).
concept(seen(_),interpretation_model,cwa,true).

operator(do_navigate(X,Y,Z),
    (
        can_traverse(X,Y,Z), available(X), at(X,Y),
        visible(Y,Z)),
    [at(X,Y)],
    [at(X,Z),seen(Y),seen(Z)],
    1).

operator(
    sample_soil(X,S,P),
    (
        at(X,P),
        at_soil_sample(P), equipped_for_soil_analysis(X),
        store_of(S,X), empty(S)),
    [empty(S), at_soil_sample(P)],
    [full(S), have_soil_analysis(X,P)],
    1).

operator(sample_rock(X,S,P),
    (
        at(X,P),
        at_rock_sample(P), equipped_for_rock_analysis(X),
        store_of(S,X), empty(S)),
    [empty(S), at_rock_sample(P)],
    [full(S), have_rock_analysis(X,P)],
    1).

operator(drop(X,Y),
    (
         store_of(Y,X), full(Y)),
    [full(Y)],
    [empty(Y)],
    1).

operator(calibrate(R,I,T,W),
    (
        equipped_for_imaging(R), calibration_target(I,T),
        at(R,W), visible_from(T,W), on_board(I,R)    ),
    [],
    [calibrated(I,R)],
    1).
```

```
operator(take_image(R,P,O,I,M),
    (
        calibrated(I,R), on_board(I,R),
        equipped_for_imaging(R), supports(I,M),
        visible_from(O,P), at(R,P)),
    [calibrated(I,R)],
    [have_image(R,O,M)],
    1).

operator(communicate_soil_data(R,L,P,X,Y),
    (
        at(R,X), at_lander(L,Y),
        have_soil_analysis(R,P), visible(X,Y),
        available(R), channel_free(L)),
    [available(R), channel_free(L)],
    [
        channel_free(L),communicated_soil_data(P),
        available(R)],
    1).

operator(communicate_rock_data(R,L,P,X,Y),
    (
        at(R,X), at_lander(L,Y),
        have_rock_analysis(R,P), visible(X,Y),
        available(R), channel_free(L)),
    [available(R), channel_free(L)],
    [
        channel_free(L),communicated_rock_data(P),
        available(R)],
    1).

operator(communicate_image_data(R,L,O,M,X,Y),
    (
        at(R,X),at_lander(L,Y),have_image(R,O,M),
        visible(X,Y), available(R), channel_free(L)),
    [available(R), channel_free(L)],
    [
        channel_free(L),
        communicated_image_data(O), available(R)],
    1).

operator(Visit(Waypoint),
    true,
    [],
    [visited(Waypoint)],
    1).
```

```
178
operator(unvisit(Waypoint),
    true,
    [visited(Waypoint)],
    [],
183 1).

operator(sense(percept(oracle,_,_)),
    true,
    [],
188 [],
    10).

operator(sense(percept(perception,
query(can_traverse(R,X,_Y),_,_),_Response)),
193 at(R,X),
    [],
    [],
    10).

198 operator(sense(percept(perception,
query(visible(X,_Y),_,_),_Response)),
    at(_R,X),
    [],
    [],
203 10).

operator(sense(percept(perception,
query(at_soil_sample(Waypoint),_,_),_Response)),
    at(_R,Waypoint),
208 [],
    [],
    10).

operator(sense(percept(perception,
213 query(at(_,_),_,_),_Response)),
    true,
    [],
    [],
    10).
218
operator(sense(percept(perception,
query(at_lander(_,_),_,_),_Response)),
    true,
    [],
223 [],
```

```
    10).

operator(sense(percept(perception,
query(channel_free(_),_,_),_Response)),
    true,
    [],
    [],
    10).

operator(sense(percept(perception,
query(available(_),_,_),_Response)),
    true,
    [],
    [],
    10).

operator(sense(percept(perception,
query(store_of(_,_),_,_),_Response)),
    true,
    [],
    [],
    10).

operator(sense(percept(perception,
query(empty(_),_,_),_Response)),
    true,
    [],
    [],
    10).

operator(sense(percept(perception,
query(equipped_for_soil_analysis(_),_,_),_Response)),
    true,
    [],
    [],
    10
).

operator(sense(percept(perception,
query(equipped_for_rock_analysis(_),_,_),_Response)),
    true,
    [],
    [],
    10
).
```

```
operator(sense(percept(perception,
    query(equipped_for_imaging(_),_,_),_Response)),
    true,
    [],
    [],
    10
).

operator(sense(percept(perception,
query(on_board(_,_),_,_),_Response)),
    true,
    [],
    [],
    10
).

operator(sense(percept(perception,
query(calibration_target(_,_),_,_),_Response)),
    true,
    [],
    [],
    10
).

operator(sense(percept(perception,
query(supports(_,_),_,_),_Response)),
    true,
    [],
    [],
    10
).

operator(sense(percept(perception,
query(waypoint(_),_,_),_Response)),
    true,
    [],
    [],
    10).


method(empty_store(S,_),
    empty(S),
    [],
    nothing,
    0).
```

```
method(empty_store(S,Rover),
    full(S),
    [drop(Rover,S)],
    do,
    0.1).

method(navigate(Rover,To),
    at(Rover,From),
    [navigate(Rover,[From],From,To)],
    do1,
    1).

method(navigate(Rover,_,_,To),
    at(Rover,To),
    [],
    nothing,
    0).

method(navigate(Rover,_,From,To),
    can_traverse(Rover,From,To),
    [do_navigate(Rover,From,To)],
    do2,
    100).

method(navigate(Rover,Visited,From,To),
    (
        can_traverse(Rover,From,Mid), Mid notin Visited,
        neg seen(Mid)),
    [
        do_navigate(Rover,From,Mid),
        navigate(Rover,[Mid|Visited],Mid,To)],
    do3,
    200000).

method(navigate(Rover,Visited,From,To),
    (
        can_traverse(Rover,From,Mid),
        seen(Mid), Mid notin Visited),
    [
        do_navigate(Rover,From,Mid),
        navigate(Rover,[Mid|Visited],Mid,To)],
    do3,
    10000000).

method(send_soil_data(Rover,Waypoint),
    (
```

```
        at_lander(L,Y),
        visible(X,Y)),
    [
        navigate(Rover,X),
        communicate_soil_data(Rover,L,Waypoint,X,Y)],
    do1,
    10).

method(get_soil_data(Waypoint),
    (
        have_soil_analysis(Rover,Waypoint)),
    [send_soil_data(Rover,Waypoint)],
    default_name,
    1).

method(get_soil_data(Waypoint),
    (
        store_of(S,Rover),
        equipped_for_soil_analysis(Rover)),
    [
        navigate(Rover,Waypoint),
        empty_store(S,Rover),
        sample_soil(Rover,S,Waypoint),
        send_soil_data(Rover,Waypoint)],
    default_name,
    1000).

method(send_rock_data(Rover,Waypoint),
    (
        at_lander(L,Y),
        visible(X,Y)),
    [
        navigate(Rover,X),
        communicate_rock_data(Rover,L,Waypoint,X,Y)],
    do1,
    10).

method(get_rock_data(Waypoint),
    (
        equipped_for_rock_analysis(Rover),
        store_of(S,Rover)),
    [
        navigate(Rover,Waypoint),
        empty_store(S,Rover),
        sample_rock(Rover,S,Waypoint),
        send_rock_data(Rover,Waypoint)
```

```
    ],
    do1,
    100).

method(
    send_image_data(Rover,Objective,Mode),
    (
        at_lander(L,Y),
        visible(X,Y)),
    [navigate(Rover,X),
    communicate_image_data(Rover,L,Objective,Mode,X,Y)],
    do1,
    10).

method(
    get_image_data(Objective,Mode),
    (
        equipped_for_imaging(Rover),
        on_board(Camera,Rover),
        supports(Camera,Mode),
        visible_from(Objective,Waypoint)),
    [
        calibrate(Rover,Camera),
        navigate(Rover,Waypoint),
        take_image(Rover,Waypoint,Objective,Camera,Mode),
        send_image_data(Rover,Objective,Mode)],
    do1,
    100).

method(
    calibrate(Rover,Camera),
    (
        calibration_target(Camera,Objective),
        visible_from(Objective,Waypoint)),
    [
        navigate(Rover,Waypoint),
        calibrate(Rover,Camera,Objective,Waypoint)],
    do1,
    5).

method(determine(percept(perception,
query(can_traverse(R,X,Y),I,C),Response)),
    at(R,X),
    [
        sense(percept(perception,
        query(can_traverse(R,X,Y),I,C),Response))],
```

```
    dc1,
    50).

method(determine(percept(perception,
query(can_traverse(R,X,Y),I,C),Response)),
    true,
    [
        navigate(R,X),
        sense(percept(perception,
            query(can_traverse(R,X,Y),I,C),Response))],
    dc1,
    500).

method(determine(percept(perception,
query(visible(W1,W2),I,C),Response)),
    (
        at(_,W1), oe visible(W1,W2), visible(W1,W2) notin I,
        call(acog_reasoner::
            not_tried_without_gain_as(
            [ac(perception,visible(W1,W2))])) ),
    [
        sense(percept(perception,
            query(visible(W1,W2),I,C),Response))],
    dc1,
    50).

method(determine(percept(perception,
query(visible(W1,W2),I,C),Response)),
    (
        at(_,W2), oe visible(W1,W2), visible(W1,W2) notin I,
        call(acog_reasoner::not_tried_without_gain_as(
            [ac(perception,visible(W1,W2))]))),
    [sense(percept(perception,
        query(visible(W1,W2),I,C),Response))],
    dc1,
    50).

method(determine(percept(perception,
query(visible(W1,W2),I,C),Response)),
    (
        can_traverse(R,_,W1), oe visible(W1,W2),
        visible(W1,W2) notin I,
        call(acog_reasoner::not_tried_without_gain_as(
            [ac(perception,visible(W1,W2))]))),
    [
        navigate(R,W1),
```

```
        sense(percept(perception,
            query(visible(W1,W2),I,C),Response))],
    dc1,
    500).

method(determine(percept(perception,query(Q,I,C),Response)),
    (
        approach_percept_query(Q,W),
        at(_,W)),
    [sense(percept(perception,query(Q,I,C),Response))],
    dc1,
    50).

method(determine(percept(perception,query(Q,I,C),Response)),
        (approach_percept_query(Q,W),
        neg at(_,W)),
    [
        navigate(_R,W),
        sense(percept(perception,query(Q,I,C),Response))],
    dc1,
    50000).

method(determine(percept(perception,
query(at(A,B),I,C),Response)),
    true,
    [sense(percept(perception,query(at(A,B),I,C),Response))],
    dc1,
    5).

method(determine(percept(perception,
query(channel_free(A),I,C),Response)),
    true,
    [
        sense(percept(perception,
            query(channel_free(A),I,C),Response))],
    dc1,
    10).

method(determine(percept(perception,
query(available(X),I,C),Response)),
    true,
    [
        sense(percept(perception,
            query(available(X),I,C),Response))],
    dc1,
    10).
```

```
method(determine(percept(perception,
query(store_of(X,Y),I,C),Response)),
    true,
    [
        sense(percept(perception,
            query(store_of(X,Y),I,C),Response))],
    dc1,
    10).

method(determine(percept(perception,
query(empty(X),I,C),Response)),
    true,
    [
        sense(percept(perception,
            query(empty(X),I,C),Response))],
    dc1,
    10).

method(determine(percept(perception,
query(equipped_for_soil_analysis(X),I,C),Response)),
    true,
    [
        sense(percept(perception,
        query(equipped_for_soil_analysis(X),I,C),Response))],
    dc1,
    10).

method(determine(percept(perception,
query(equipped_for_rock_analysis(X),I,C),Response)),
        true,
    [
        sense(percept(perception,query(
            equipped_for_rock_analysis(X),I,C),Response))],
    dc1,
    10).

method(determine(percept(perception,
query(equipped_for_imaging(X),I,C),Response)),
        true,
    [
        sense(percept(perception,
            query(equipped_for_imaging(X),I,C),Response))
    ],
    dc1,
    10).
```

```
593 method(determine(percept(perception,
query(on_board(X,Y),I,C),Response)),
    true,
    [
        sense(percept(perception,
598             query(on_board(X,Y),I,C),Response))
    ],
    dc1,
    10).

603 method(determine(percept(perception,
query(calibration_target(X,Y),I,C),Response)),
    true,
    [
        sense(percept(perception,
608             query(calibration_target(X,Y),I,C),Response))
    ],
    dc1,
    10).

613 method(determine(percept(perception,
query(supports(X,Y),I,C),Response)),
        true,
    [
        sense(percept(perception,
618             query(supports(X,Y),I,C),Response))],
    dc1,
    10).

method(determine(percept(perception,
623     query(at_lander(X,Y),I,C),Response)),
        true,
    [
        sense(percept(perception,
            query(at_lander(X,Y),I,C),Response))],
628     dc1,
    10).

method(determine(percept(perception,
    query(waypoint(X),I,C),Response)),
633         true,
    [
        sense(percept(perception,
            query(waypoint(X),I,C),Response))],
    dc1,
```

```
638      10).

:- end_object.
```

# Acknowledgments

First, I would like to thank my advisor Prof. Jianwei Zhang for giving me visions, helping me to enter the research community and making this work possible by providing an excellent research environment. Moreover, I thank Prof. Bernd Neumann for inspiring discussions and helpful Feedback.

I will definitely miss all my CINACS colleagues including Sabrina Boll, Stephanie Badde, Niels Beuck, Christopher Baumgärtner, Kris Lohmann, Mario Maiworm, Sebastian Gluth, Jens Kleesiek, Johannes Bauer, Christian Floß and Jianhua Zhang. I will never forget the awesome time we had. We had so much fun in Beijing. Special thanks goes to the great Chinese food and "The Stairs".

Special thanks goes to Felix Lindner for reading initial drafts and providing helpful comments and criticism.

I thank Martin Weser and Sascha Jockel for giving me valuable feedback and helping me to establish myself in a new research environment.

I would like to thank all my colleagues from the TAMS group for providing professional support, for valuable discussions and for a nice working atmosphere. Specifically I want to thank Lu, Norman, Andreas, Bernd, Hannes, Dennis, Mohammed, Ben, Junhao Xiao, Guoyuan Li and Gang Cheng. I thank Heye Vöcking, Pulpo and Emil for their support.

Finally, I would like to thank Anna for supporting and encouraging me while I was writing this thesis and for reminding me that work is not everything in life.

This work is funded by the DFG German Research Foundation (grant 1247) – International Research Training Group CINACS (Cross-modal Interactions in Natural and Artificial Cognitive Systems).

# List of Figures

# List of Tables

# Index

# Publications

The work presented in this thesis has led to several publications. Below is a list of all prior publications:

Off, D. and Zhang, J. (2012). Continual HTN planning and acting in open-ended domains: Considering knowledge acquisition opportunities. *In Proceedings of International Conference on Agents and Artificial Intelligence (ICAART)*.

Off, D. and Zhang, J. (2011a). Continual HTN robot task planning in open-ended domains: A case study. *In Proceedings of the AAAI-11 Workshop on Automated Action Planning for Autonomous Mobile Robots (PAMR)*.

Off, D. and Zhang, J. (2011b). Multimodal integration processes in plan-based service robot control. *Tsinghua Science and Technology*, 16(1):1 – 6.

Off, D. and Zhang, J. (2011c). Open-ended domain model for continual forward search HTN planning. *In Proceedings of the ICAPS-2011 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Weser, M., Off, D., and Zhang, J. (2010). HTN robot planning in partially observable dynamic environments. *In Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1505–1510.

# Bibliography


Ambros-Ingerson, J. A. and Steel, S. (1988). Integrating planning, execution and monitoring. In *AAAI*, pages 83–88.

Arkin, R. C. (1998). *Behavior-based Robotics*. MIT Press, 2 edition.

Asfour, T., Regenstein, K., Azad, P., Schröder, J., Bierbaum, A., Vahrenkamp, N., and Dillmann, R. (2006). ARMAR-III: An integrated humanoid platform for sensory-motor control. In *IEEE-RAS International Conference on Humanoid Robots*.

Au, T.-C. and Nau, D. S. (2006). The incompleteness of planning with volatile external information. In *ECAI*, pages 839–840.

Au, T.-C., Nau, D. S., and Subrahmanian, V. S. (2004). Utilizing volatile external information during planning. In *ECAI*, pages 647–651.

Bäckström, C. (1992). Equivalence and tractability results for sas+ planning. In *KR*, pages 126–137.

Bäckström, C. and Klein, I. (1991). Planning in polynomial time: the sas-pubs class. *Computational Intelligence*, 7:181–197.

Bäckström, C. and Nebel, B. (1995). Complexity results for sas+ planning. *Computational Intelligence*, 11:625–656.

Baier-Löwenstein, T. (2008). *Lernen der Handhabung von Alltagsgegenständen im Kontext eines Service-Roboters*. PhD thesis, Universität Hamburg, Von-Melle-Park 3, 20146 Hamburg.

Baral, C., Kreinovich, V., and Trejo, R. (2000). Computational complexity of planning and approximate planning in the presence of incompleteness. *Artif. Intell.*, 122(1-2):241–267.

Beetz, M. (2001). Structured reactive controllers. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):25–55.

Beetz, M. (2002a). *Plan-based control of robotic agents: improving the capabilities of autonomous robots*. Springer-Verlag, Berlin, Heidelberg.

Beetz, M. (2002b). Plan representation for robotic agents. In Ghallab et al. (2002), pages 223–232.

Ben-Ari, M. (2001). *Mathematical Logic for Computer Science*. Springer.

Brachman, R. J. and Levesque, H. J. (2004). *Knowledge Representation and Reasoning*. Morgan Kaufman Publ Inc.

Brenner, M. and Nebel, B. (2009). Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–21.

Brooks, R. A. (1991). Intelligence without representation. *Artif. Intell.*, 47(1-3):139–159.

Burgard, W., Cremers, A. B., Fox, D., Hähnel, D., Lakemeyer, G., Schulz, D., Steiner, W., and Thrun, S. (1999). Experiences with an interactive museum tour-guide robot. *Artif. Intell.*, 114(1-2):3–55.

Cadoli, M. and Donini, F. M. (1997). A survey on knowledge compilation. *AI Commun.*, 10(3-4):137–150.

Chalupsky, H., MacGregor, R. M., and Russ, T. (2010). *PowerLoom Manual (Version 1.48)*. University of Southern California, Information Sciences Institute.

Chalupsky, H. and Russ, T. A. (2002). Whynot: Debugging failed queries in large knowledge bases. In *AAAI/IAAI*, pages 870–877.

Chien, S., Rabideau, G., Knight, R., Sherwood, R., Engelhardt, B., Mutz, D., Estlin, T., Smith, B., Fisher, F., Barrett, T., Stebbins, G., and Tran, D. (2000). Aspen - automated planning and scheduling for space mission operations. In *in Space Ops*.

Clark, K. L. (1987). Negation as failure. *Logic and databases*, pages 293–322.

Darwiche, A. and Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research (JAIR)*, 17:229–264.

Davis, R., Shrobe, H. E., and Szolovits, P. (1993). What is a knowledge representation? *AI Magazine*, 14(1):17–33.

Deransart, P., Eb-Dbali, A., and Cervoni, L. (1996). *Prolog: The Standard*. Springer.

Doan, A. and Haddawy, P. (1995). Decision-theoretic refinement planning: Principles and application. Technical Report TR-95-01-01, Unversity of Wisconsin-Milwaukee.

Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. (2009). Semantic attachments for domain-independent planning systems. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 114–121. AAAI Press.

Drummond, M. and Currie, K. (1989). Goal ordering in partially ordered plans. In *IJCAI*, pages 960–965.

Erol, K., Hendler, J. A., and Nau, D. S. (1994). Umcp: A sound and complete procedure for hierarchical task-network planning. In Hammond (1994), pages 249–254.

Estlin, T., Castano, R., Anderson, R., Gaines, D., Fisher, F., and Judd, M. (2003). Learning and planning for mars rover science. In *In Proc. of IJCAI Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World Modeling, Planning, Learning, and Communicating*. Morgan Kaufmann Publishers.

Estlin, T., Fisher, F., Gaines, D., Chouinard, C., Schaffer, S., and Nesnas, I. (2002). Continuous planning and execution for a mars rover. In *International NASA Workshop on Planning and Scheduling for Space*.

Etzioni, O., Golden, K., and Weld, D. S. (1997). Sound and efficient closed-world reasoning for planning. *Artif. Intell.*, 89(1-2):113–148.

Etzioni, O., Hanks, S., Weld, D. S., Draper, D., Lesh, N., and Williamson, M. (1992). An approach to planning with incomplete information. In *KR*, pages 115–125.

Fagin, R., Halpern, J. Y., Moses, Y., and Yardi, M. Y. (1995). *Reasoning About Knowledge*. MIT Press.

Ferrein, A. and Lakemeyer, G. (2008). Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems*, 56(11):980–991.

Fikes, R., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artif. Intell.*, 3(1-3):251–288.

Fikes, R. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208.

Fox, D. and Gomes, C. P., editors (2008). *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*. AAAI Press.

Fox, M. and Long, D. (1995). Hierarchical planning using abstraction. *Control Theory and Applications*, 142(3):197–210.

Galindo, C., Fernandez-Madrigal, J.-A., González, J., and Saffiotti, A. (2008). Robot task planning using semantic maps. *Robotics and Autonomous Systems*, 56(11):955–966.

Galindo, C., FernÃ¡ndez-Madrigal, J.-A., GonzÃ¡lez, J., and Saffiotti, A. (2007). Using semantic information for improving efficiency of robot task planning. In *in: ICRA Workshop: Semantic Information in Robotics*.

Galindo, C., Saffiotti, A., Coradeschi, S., and Buschka, P. (2005). Multi-hierarchical semantic maps for mobile robotics. In *in Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, 2005*, pages 3492–3497.

Ghallab, M., Hertzberg, J., and Traverso, P., editors (2002). *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, April 23-27, 2002, Toulouse, France*. AAAI.

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). Pddl - the planning domain definition language: Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning Theory and Practice*. Elsevier Science.

Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2000). Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169.

Giacomo, G. D. and Levesque, H. J. (1999). An incremental interpreter for high-level programs with sensing. In Levesque, H. J. and Pirri, F., editors, *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 86–102. Springer, Berlin.

Goebelbecker, M., Gretton, C., and Dearden, R. (2011). A switching planner for combined task and observation planning. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence (AAAI 2011)*.

Golden, K. (1998). Leap before you look: Information gathering in the puccini planner. In *AIPS*, pages 70–77.

Grounds, M. and Kudenko, D. (2007). Combining reinforcement learning with symbolic planning. In Tuyls, K., Nowé, A., Guessoum, Z., and Kudenko, D., editors, *Adaptive Agents and Multi-Agents Systems*, volume 4865 of *Lecture Notes in Computer Science*, pages 75–86. Springer.

Hammond, K. J., editor (1994). *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems, University of Chicago, Chicago, Illinois, USA, June 13-15, 1994*. AAAI.

Hartanto, R. (2009). PhD thesis, Osnabrück University.

Hartanto, R. and Hertzberg, J. (2008). Fusing dl reasoning with htn planning. In *KI*, pages 62–69.

Hartanto, R. and Hertzberg, J. (2009). On the benefit of fusing dl-reasoning with htn-planning. In *KI*, pages 41–48.

Hayashi, H., Cho, K., and Ohsuga, A. (2004). A new htn planning framework for agents in dynamic environments. In *CLIMA IV*, pages 108–133.

Helmert, M. and Röger, G. (2008). How good is almost perfect? In Fox and Gomes (2008), pages 944–949.

Helwig, J. and Haddawy, P. (1996). Abstraction-based approach to interleaving planning and execution in partially-observable domains. In *AAAI Fall Symposium*.

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with hexq. In Sammut, C. and Hoffmann, A. G., editors, *ICML*, pages 243–250. Morgan Kaufmann.

Hertzberg, J. and Chatila, R. (2008). Ai reasoning methods for robotics. In *Springer Handbook of Robotics*, chapter 9, pages 207–223. Springer.

Hoffmann, J. and Brafman, R. (2006). Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6–7):507–541.

Hogg, C., Muñoz-Avila, H., and Kuter, U. (2008). Htn-maker: Learning htns with minimal additional knowledge engineering required. In Fox and Gomes (2008), pages 950–956.

Ilghami, O. (2006). Documentation for JSHOP2.

Ilghami, O., Muñoz-Avila, H., Nau, D. S., and Aha, D. W. (2005). Learning approximate preconditions for methods in hierarchical plans. In *ICML*, pages 337–344.

Ilghami, O., Nau, D. S., and Muñoz-Avila, H. (2006). Learning to do htn planning. In *ICAPS*, pages 390–393.

Ilghami, O., Nau, D. S., Muñoz-Avila, H., and Aha, D. W. (2002). Camel: Learning method preconditions for htn planning. In Ghallab et al. (2002), pages 131–142.

Jockel, S. (2009). *Crossmodal Learning and Prediction of Autobiographical Episodic Experiences using a Sparse Distributed Memory*. PhD thesis, University of Hamburg.

Jonsson, P. and Bäckström, C. (1998). State-variable planning under structural restrictions: Algorithms and complexity. *Artif. Intell.*, 100(1-2):125–176.

Kaelbling, L. P. and Lozano-Pérez, T. (2011). Hierarchical task and motion planning in the now. In *ICRA*, pages 1470–1477.

Keller, T., Eyerich, P., and Nebel, B. (2010). Task planning for an autonomous service robot. In Dillmann, R., Beyerer, J., Hanebeck, U. D., and Schultz, T., editors, *KI*, volume 6359 of *Lecture Notes in Computer Science*, pages 358–365. Springer.

Knoblock, C. A. (1995). Planning, executing, sensing, and replanning for information gathering. In *IJCAI*, pages 1686–1693.

Koenig, S. (2010). Creating a uniform framework for task and motion planning: A case for incremental heuristic search? In *Combining Action and Motion Planning Workshop, International Conference on Automated Planning and Scheduling (ICAPS)*.

Koenig, S. and Likhachev, M. (2002). Improved fast replanning for robot navigation in unknown terrain. In *ICRA*, pages 968–975. IEEE.

Kortenkamp, D. and Simmons, R. G. (2008). Robotic systems architectures and programming. In Siciliano, B. and Khatib, O., editors, *Springer Handbook of Robotics*, chapter 8, pages 187–206. Springer.

Kraft, D., Başeski, E., Popović, M., Batog, A. M., Kjær-Nielsen, A., Krüger, N., Petrick, R., Geib, C., Pugeault, N., Steedman, M., Asfour, T., Dillmann, R., Kalkan, S., Wörgötter, F., Hommel, B., Detry, R., and Piater, J. (2008). Exploration and planning in a three-level cognitive architecture. In *International Conference on Cognitive Systems (Workshop at the IEEE International Conference on Robotics and Automation)*. Extended Abstract.

Kuter, U., Nau, D., Reisner, E., and Goldman, R. (2007). Conditionalization: Adapting forward-chaining planners to partially observable environments. In *Proceedings of the ICAPS-07 Workshop on Planning and Plan Execution for Real-World Systems – Principles and Practices for Planning in Execution*.

Kuter, U., Sirin, E., Nau, D. S., Parsia, B., and Hendler, J. A. (2004). Information gathering during planning for web service composition. In *International Semantic Web Conference*, pages 335–349.

Lakemeyer, G. (1998). On sensing and offline interpreting in golog. Technical report, Department of Computer Science, Aachen University of Technology.

Latombe, J.-C. (2003). *Robot motion planning*. Kluwer.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

Levesque, H. and Lakemeyer, G. (2008). Cognitive robotics. In van Harmelen, F., Lifschitz, V., and Porter, B., editors, *Handbook of Knowledge Representation*, chapter 23, pages 869–886. Elsevier Science.

Levesque, H. and Reiter, R. (1998). Beyond planning. In *In AAAI Spring Symposium on Integrating Robotics Research*.

Levesque, H. J. (1996). What is planning in the presence of sensing? In *AAAI/IAAI, Vol. 2*, pages 1139–1146.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. (1997). Golog: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83.

Littman, M. L., Goldsmith, J., and Mundhenk, M. (1998). The computational complexity of probabilistic planning. *J. Artif. Intell. Res. (JAIR)*, 9:1–36.

Lloyd, J. and Hayward, V. (1992). *Multi-RCCL User's Guide*. McGill University.

Long, D. and Fox, M. (2003). The 3rd international planning competition: results and analysis. *J. Artif. Int. Res.*, 20(1):1–59.

Luo, R. C. and Kay, M. G., editors (1995). *Multisensor integration and fusion for intelligent machines and systems*. Ablex Publishing Corp., Norwood, NJ, USA.

Marthi, B. (2006). *Concurrent Hierarchical Reinforcement Learning*. PhD thesis, University of California, Berkeley.

Marthi, B., Russell, S. J., and Wolfe, J. (2007). Angelic semantics for high-level actions. In *ICAPS*, pages 232–239.

Marthi, B., Russell, S. J., and Wolfe, J. (2008). Angelic hierarchical planning: Optimal and online algorithms. In *ICAPS*, pages 222–231.

McCluskey, T. L. (2002). Knowledge engineering: Issues for the ai planning community. In *Proceedings of the AIPS-2002 Workshop on Knowledge Engineering Tools and Techniques for AI Planning*.

McDermott, D. (1992). Robot planning. *AI Magazine*, 13:55–79.

McGovern, A. and Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In Brodley, C. E. and Danyluk, A. P., editors, *ICML*, pages 361–368. Morgan Kaufmann.

Mitsubishi heavy industries, L. (2002). *General Purpose Robot PA10 Series Programming Manual*. Mitsubishi heavy industries, LTD., rev.3 edition.

Moura, P. (2003). *Logtalk Design of and Object-Oriented Logic Programming Language*. PhD thesis, University of Beira Interior.

Müller, A. (2008). *Transformational Planning for Autonomous Household Robots using Libraries of Robust and Flexible Plans*. PhD thesis, Technische Universität München.

Müller, A. and Beetz, M. (2007). Towards a plan library for household robots. In *Proceedings of the ICAPS'07 Workshop on Planning and Plan Execution for Real-World Systems: Principles and Practices for Planning in Execution*, Providence, USA.

Murphy, R. R. (2000). *Introduction to AI robotics*. Intelligent robotics and autonomous agents. MIT Press. A Bradford book.

Nau, D., Au, T. C., Ilghami, O., Kuter, U., Murdock, W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN planning system. *Journal on Artificial Intelligence Research*, 20.

Nau, D. S. (2007). Current trends in automated planning. *AI Magazine*, 28(4):43–58.

Nau, D. S., Cao, Y., Lotem, A., and Muñoz-Avila, H. (1999). Shop: Simple hierarchical ordered planner. In *IJCAI*, pages 968–975.

Nejati, N., Langley, P., and Konik, T. (2006). Learning hierarchical task networks by observation. In *International Conference on Machine Learning . Pittsburgh, PA.*, pages 665–672.

Nguyen, N. T. (2008). *Advanced Methods for Inconsistent Knowledge Management*. Springer.

Nilsson, N. J. (1984). Shakey the robot. Technical Report 323, SRI.

Nilsson, U. and Maluszynski, J. (1995). *Logic, Programming, and PROLOG*. John Wiley & Sons, Inc., New York, NY, USA.

Off, D. and Zhang, J. (2011a). Continual HTN robot task planning in open-ended domains: A case study. In *Proceedings of the AAAI-11 Workshop on Automated Action Planning for Autonomous Mobile Robots (PAMR)*.

Off, D. and Zhang, J. (2011b). Multimodal integration processes in plan-based service robot control. *Tsinghua Science and Technology*, 16(1):1 – 6.

Off, D. and Zhang, J. (2011c). Open-ended domain model for continual forward search HTN planning. In *Proceedings of the ICAPS-2011 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Off, D. and Zhang, J. (2012). Continual HTN planning and acting in open-ended domains: Considering knowledge acquisition opportunities. In *Proceedings of International Conference on Agents and Artificial Intelligence (ICAART)*.

Pednault, E. P. D. (1988). Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4:356–372.

Pednault, E. P. D. (1989). Adl: Exploring the middle ground between strips and the situation calculus. In *KR*, pages 324–332.

Petrick, R. P. A. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In Ghallab et al. (2002), pages 212–222.

Petrick, R. P. A. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In Dubois, D., Welty, C. A., and Williams, M.-A., editors, *KR*, pages 613–622. AAAI Press.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, illustrated edition edition.

Rintanen, J. (1999). Constructing conditional plans by a theorem-prover. *J. Artif. Intell. Res. (JAIR)*, 10:323–352.

Rintanen, J. (2004). Complexity of planning with partial observability. In *ICAPS*, pages 345–354.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Sacerdoti, E. (1975). The nonlinear nature of plans. Technical Report A726854, SRI.

Sardiña, S., Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2004). On the semantics of deliberation in indigolog - from theory to implementation. *Ann. Math. Artif. Intell.*, 41(2-4):259–299.

Scherer, T. (2004). *A Mobile Service Robot for Automisation of Sample Taking and Sample Management in a Biotechnological Pilot Laboratory*. PhD thesis, University of Bielefeld.

Shanahan, M. (2000). Reinventing shakey. In Minker, editor, *Logic-Based Artificial Intelligence*, chapter 11, pages 233–253. Kluwer Academic.

Shanahan, M. and Witkowski, M. (2001). High-level robot control through logic. In Castelfranchi, C. and Lespérance, Y., editors, *Intelligent Agents VII Agent Theories Architectures and Languages*, volume 1986 of *Lecture Notes in Computer Science*, pages 302–308. Springer Berlin / Heidelberg.

Simmons, R. G., Goodwin, R., Haigh, K. Z., Koenig, S., O'Sullivan, J., and Veloso, M. M. (1997). Xavier: experience with a layered robot architecture. *SIGART Bulletin*, 8(1-4):22–33.

Simsek, Ö. and Barto, A. G. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. In Brodley, C. E., editor, *ICML*, volume 69 of *ACM International Conference Proceeding Series*. ACM.

Stentz, A. (1995). The focussed d* algorithm for real-time replanning. In *IJCAI*, pages 1652–1659.

Sterling, L. and Shapiro, E. (1994). *The Art of Prolog: Advanced Programming Techniques*. The MIT Press.

Subrahmanian, V. S. (1999). Nonmonotonic logic programming. *IEEE Trans. Knowl. Data Eng.*, 11(1):143–152.

Talamadupula, K., Benton, J., Kambhampati, S., Schermerhorn, P. W., and Scheutz, M. (2010a). Planning for human-robot teaming in open worlds. *ACM TIST*, 1(2):14.

Talamadupula, K., Benton, J., Schermerhorn, P. W., Kambhampati, S., and Scheutz, M. (2010b). Integrating a closed world planner with an open world robot: A case study. In *AAAI*.

Tam, K., Lloyd, J., Lespérance, Y., Levesque, H. J., Lin, F., Marcu, D., Reiter, R., and Jenkin, M. R. M. (1997). Controlling autonomous robots with *golog*. In *Australian Joint Conference on Artificial Intelligence*, pages 1–12.

Tate, A. (1977). Generating project networks. In *IJCAI*, pages 888–893.

Tate, A. and Dalton, J. (2003). O-plan: a common lisp planning web service. In *Proceedings of the International Lisp Conference*.

Tenorth, M., Nyga, D., and Beetz, M. (2010). Understanding and Executing Instructions for Everyday Manipulation Tasks from the World Wide Web. In *IEEE International Conference on Robotics and Automation (ICRA)*.

Thiébaux, S., Hoffmann, J., and Nebel, B. (2005). In defense of pddl axioms. *Artificial Intelligence*, 168(1-2):38–69.

Thielscher, M. (1998). Introduction to the fluent calculus. *Electron. Trans. Artif. Intell.*, 2:179–192.

Thielscher, M. (2000). Representing the knowledge of a robot. In *KR*, pages 109–120.

Thielscher, M. (2005a). FLUX: A logic programming method for reasoning agents. *Theory Pract. Log. Program.*, 5:533–565.

Thielscher, M. (2005b). *Reasoning robots : the art and science of programming robotic agents*. Springer.

Thrun, S., Bennewitz, M., Burgard, W., Cremers, A., Dellaert, F., Fox, D., Hähnel, D., Rosenberg, C., Roy, N., Schulte, J., and Schulz, D. (1999). MINERVA: A second generation mobile tour-guide robot.

To, S. T., Son, T. C., and Pontelli, E. (2011). Contingent planning as and/or forward search with disjunctive representation. In *ICAPS*.

Townsend, W. (2000). The barretthand grasper – programmably flexible part handling and assembly. *Industrial Robot: An International Journal*, 22:3:181–188.

Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending graphplan to handle uncertainty & sensing actions. In *AAAI/IAAI*, pages 897–904.

Weser, M. (2010). *Hierarchical Memory Organization of Multimodal Robot Skills for Plan-based Robot Control*. PhD thesis, University of Hamburg.

Weser, M., Off, D., and Zhang, J. (2010). HTN robot planning in partially observable dynamic environments. In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1505–1510.

Westhoff, D., Stanek, H., and Zhang, J. (2006). Distributed applications for robotic systems using roblet-technology. In *Proceedings of the 37th International Symposium on Robotics and Deutsche Fachtagung Robotik 2006*, Munich, Germany. VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik.

Wilkins, D. E. (1983). Representation in a domain-independent planner. In *IJCAI*, pages 733–740.

Wilkins, D. E. (1990). Can ai planners solve practical problems? *Computational Intelligence*, 6:232–246.

Winograd, T. (1972). *Understanding Natural Language*. Academic Press.

Wolfe, J., Marthi, B., and Russell, S. J. (2010). Combined task and motion planning for mobile manipulation. In *ICAPS*, pages 254–258.

Wyrobek, K. A., Berger, E. H., der Loos, H. F. M. V., and Salisbury, J. K. (2008). Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In *ICRA*, pages 2165–2170. IEEE.

Yoon, S., Fern, A., and Givan, R. (2008). Learning control knowledge for forward search planning. *J. Mach. Learn. Res.*, 9:683–718.

Young, R. M., Pollack, M. E., and Moore, J. D. (1994). Decomposition and causality in partial-order planning. In Hammond (1994), pages 188–194.