CONTINUAL HTN PLANNING AND ACTING IN OPEN-ENDED DOMAINS

Considering Knowledge Acquisition Opportunities

Dominik Off and Jianwei Zhang
TAMS, University of Hamburg, Vogt-Koelln-Strasse 30, Hamburg, Germany

Keywords: Continual planning, HTN planning, Reasoning, Knowledge representation, Plan execution.

Abstract: Generating plans in order to perform high-level tasks is difficult for agents that act in open-ended domains where it is unreasonable to assume that all necessary information is available a priori. This paper addresses this challenge by presenting a planning-based control system that is able to perform tasks in open-ended domains. The control system is based on a new HTN planning approach that additionally considers decompositions that would be applicable with respect to a consistent extension of the domain model at hand. The proposed control system constitutes a continual planning and acting system that interleaves planning and acting so that missing information can be acquired by means of active information gathering. Experimental results demonstrate that this control architecture can perform tasks in several domains even if the agent initially has no factual knowledge.

1 INTRODUCTION

If we instruct artificial agents to perform a task, then we usually want to tell them what to do, but not how to do it (e.g., in terms of a detailed sequence of low-level commands). In other words, we want agents to autonomously and flexibly plan how they can reasonably perform a given task. Planning their future course of action is particularly difficult for agents (e.g., robots) that act in a *dynamic* and *open-ended* environment where it is unreasonable to assume that a complete representation of the state of the domain is available. We define an *open-ended domain* as a domain in which an agent can in general neither be sure to have all information nor to know all possible states (e.g., all objects) of the world it inhabits.

Planning algorithms have been developed that in principle are efficient enough to solve complex 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 generate conditional plans—or policies—for all possible contingencies. Unfortunately, these approaches are computationally hard, scale badly in dynamic unstructured domains and are only applicable if it is possible to foresee all possible outcomes of a knowledge acquisition process (Rintanen, 1999; Littman et al., 1998). Therefore, these approaches can hardly be applied to the dynamic and open-ended domains we are interested in. Consider, for example, a robot agent that is instructed to bring Bob's mug into the kitchen, but does not know the location of the mug. Generating a plan for all possible locations in a three dimensional space obviously is unreasonable and practically impossible.

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. Existing continual planning systems can deal with incomplete information. However, they usually rely on the assumption that all possible states of a domain are known. 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

Off D. and Zhang J..
CONTINUAL HTN PLANNING AND ACTING IN OPEN-ENDED DOMAINS - Considering Knowledge Acquisition Opportunities.
DOI: 10.5220/0003704500160025
In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence* (ICAART-2012), pages 16-25
ISBN: 978-989-8425-95-9

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. Is 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 how 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, we propose a continual planning and acting approach that is able to deal with these kind of situations and thus can enable an agent to perform tasks in a larger set of situations.

We assume that agents are able to acquire information from external sources. The key problem we are trying to address is not how to generate a plan for a knowledge acquisition task, since planning to acquire certain information (e.g., determining whether the kitchen door is open) technically does not differ from generating plans for other tasks (e.g., making a cup of coffee). In contrast, we are trying to give an answer to the following questions: How can an agent determine knowledge acquisition activities that make it possible to find a plan when necessary information is missing? When is it more reasonable to acquire additional information prior to continuing the planning process? How to automatically switch between planning and acting?

The main contributions of this work are:

- to propose the new HTN planning system *ACogPlan* that additionally considers planning alternatives that are possible with respect to a consistent extension of the domain model at hand, and is able to autonomously decide when it is more reasonable to acquire additional information prior to continuing the planning process;
- to propose the ACogPlan based high-level control system *ACogControl* that enables an agent to perform tasks in open-ended domains;
- and to present a set of experiments that demonstrate the performance characteristics of the overall approach.

2 HTN PLANNING IN OPEN-ENDED DOMAINS

In this section we present the ACogPlan continual HTN planning system. We describe the planning phase of the overall continual planning and acting control architecture.

2.1 General Idea

The proposed planning system ACogPlan is an extension of the SHOP (Nau et al., 1999) forward search (i.e., forward decomposition) *Hierarchical Task Network (HTN)* planning system. The SHOP algorithm plans by successively choosing an instance of a *relevant*[1] HTN method or planning operator for which an instance of the precondition can be derived with respect to the domain model at hand. However, in open-ended domains 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 planning system *ACogPlan* is to also consider instances of relevant HTN methods and planning operators for which the precondition cannot be derived but might be derivable with respect to a consistent extension of the domain model (i.e., if additional information is available).

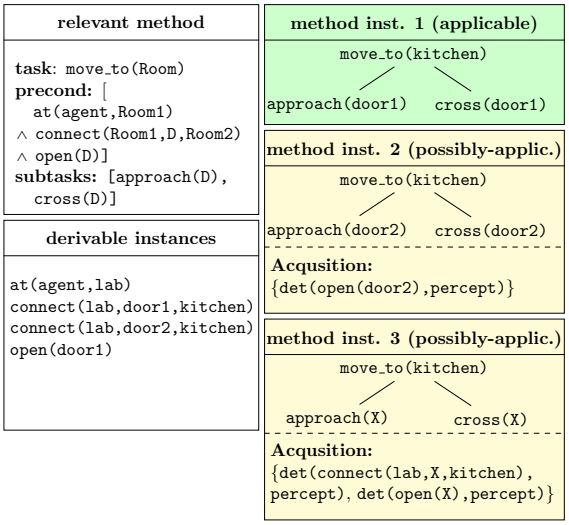

Figure 1: Applicable and possibly-applicable method instances for the task `move_to(kitchen)`.

For example, consider a simple situation where a robot is instructed to perform the task

[1] as defined in (Ghallab et al., 2004, Definition 11.4)

move_to(*kitchen*) as illustrated by Figure 1.[2] In this situation there is only one relevant HTN method. It is known that the robot is in the lab, the lab is connected to the kitchen via `door1` and `door2`, and `door1` is open. For the illustrated example, existing HTN planners would only consider the first instance of the relevant HTN method that plans to approach and cross `door1`. The proposed HTN planning algorithm *ACogPlan*, however, also considers two additional instances of the relevant HTN method which cannot directly be applied, but are applicable in a consistent extension of the given domain. 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 could 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.

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 possible-applicable method instances a planner would fail to generate a plan for the task `move_to(kitchen)` and thus 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.

2.2 Open-Ended Domain Model

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. Most existing automated planning systems are unable to do that, since their underlying domain model is based on the assumption that all information is available at the beginning of the planning process (Nau, 2007). In contrast, the proposed HTN planning system *ACogPlan* is based on the open-ended domain model *ACogDM*. ACogDM enables the planner to reason about relevant extensions of its domain model. The key concepts of ACogDM are described briefly in this section.

A planner should only consider domain model extensions that are *possible* and *relevant* with respect to the overall task. However, how can a planner infer what is relevant and possible? The domain information encoded in HTN methods can nicely be exploited in order to infer which information is relevant. A relevant method or planning operator can actually be applied if and only if its precondition p holds (i.e., an instance $p\sigma$[3] is derivable) with respect to the given domain model. Therefore, we define the set of *relevant preconditions* with respect to a given *planning context* (i.e., a domain model and a task list) to be the set of all preconditions of relevant methods or planning operators. An HTN planner cannot—except backtracking—continue the planning process in situations where no relevant precondition is derivable with respect to the domain model at hand. The notation of a relevant precondition is a first step to determine relevant extensions of a domain model, since only domain model extensions that make the derivation of an additional instance of a relevant precondition possible constitute an additional way to continue the planning process. All other possible extensions are irrelevant, because they do not imply additional planning alternatives. In other words, if it were possible to acquire additional information which implies the existence of a new instance of a relevant precondition, then the planning process could be continued in an alternative manner. As already pointed out, this is particularly relevant for situations in which it would otherwise be impossible to find any plan at all.

In order to formalize this we introduce the following concepts: a *possibly-derivable statement* (e.g., a precondition) and an *open-ended literal*. Let L_x be a set of literals and p be a precondition. p is called *possibly-derivable* iff the existence of a new instance $l\sigma$ for each $l \in L_x$ implies the existence of a new instance $p\sigma$ of p. Obviously this definition is only useful if the existence of an additional instance for each $l \in L_x$ is possible. A literal for which the existence of non-derivable instances is possible is called *open-ended*. Based on that, one can say that a possibly-derivable precondition constitutes the partition of a precondition into a derivable and an open-ended part (i.e., a set of open-ended literals).

For example, consider the situation illustrated by Figure 1. In this example there are three differ-

[2] Please note that in the context of this work variables will be written as alphanumeric identifiers beginning with capital letters.

[3] In the context of this work σ denotes a substitution.

ent situations in which the precondition of the HTN method is possibly-derivable. In all cases `Room1` is substituted with `lab` and `Room2` is substituted with `kitchen`. Furthermore, in the first situation `D` is substituted with `door1` and the precondition is possibly-derivable with respect to the agents domain model and the set of open-ended literals $\{\}$. In the second case, `D` is substituted with `door2` and the precondition is possibly-derivable with respect to the set of open-ended literals $\{$`open(door2)`$\}$. In the last case, `D` is not instantiated and the precondition is possibly-derivable with respect to the set of open-ended literals $\{$`connect(lab,D,kitchen),open(D)`$\}$. Thus, in this example the open-ended domain model ACogDM can tell the robot agent that it can cross door1, or cross door2 if it can find out that door2 is open, or cross another door `D` if it finds another door `D` that connects the lab and the kitchen and is open. In this way, ACogDM can enable a planner to reason about possible and relevant extensions of its domain model.

2.3 Planning Algorithm

In this section we present the key conceptualizations and the algorithm of the proposed planning system.

2.3.1 Preliminaries

If we want agents to acquire additional instances of a set of open-ended literals, then it should be considered that there might be dependencies between literals. For example, for the set of open-ended literals $\{mug(X), color(X, red)\}$ one cannot independently acquire an instance of $mug(X)$ and an instance of $color(X, red)$, because one needs to find an instance of X which represents a mug as well as a red object. Let l_1, l_2 be literals that are part of a precondition p in disjunctive normal form and $var(l)$ denote the set of variables of a literal l. l_1 and l_2 are called *dependent* (denoted as $l_1 \leftrightarrow l_2$) iff l_1 and l_2 are part of the same conjunctive clause and $((var(l_1) \cap var(l_2) \neq \emptyset)$ or l_1 and l_2 are identical or $(\exists_{l_3} l_1 \leftrightarrow l_3 \wedge l_3 \leftrightarrow l_2))$.

Agents (e.g., robots) can usually 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., perception), however, involves additional planning and execution. For the purpose of enabling ACogPlan to generate knowledge acquisition plans we use a particular kind of task, namely a *knowledge acquisition task*. A Knowledge acquisition task has the form $det(l, I, C, ks)$ where l is a literal, I is the set of all derivable instances of l, C is a set of literals that are dependent on l, and ks is a knowledge source. In other words, $det(l, I, C, ks)$ is the task of acquiring an instance $l\sigma$ of l from the knowledge source ks such that $l\sigma \notin I$ (i.e., $l\sigma$ is not already derivable) and for all $c \in C$ an instance of $c\sigma$ is derivable. For example, $det(open(kitchen\_door), \emptyset, \emptyset, percept)$ is the task of determining whether the kitchen door is open by means of perception. Furthermore, $det(mug(X), [mug(bobs\_mug)], [in\_room(X, r1), red(X)], hri(bob))$ constitutes the task of finding a red mug which is located in the room $r1$ and is not Bob's mug by means of human robot interaction with Bob. Like for other tasks, we can define HTN methods that describe how to perform a knowledge acquisition task. For example, Figure 2 shows a method for the acquisition task of determining whether a door is open. Every method has an expected cost that describes how expensive it is to perform a task as described by the method. In this example the cost is "hard-coded", but it is also possible to calculate a situation dependent cost.

```
method(det(open(Door),I,C,percept),
(door(Door)),
% precondition
[approach(Door),          % subtasks
sense(open(Door),percept)],
50).                      % cost
```

Figure 2: Example HTN method for an acquisition task.

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, and (3) 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 $\{(on\_table(bobs\_mug), percept), (white\_coffee(bob), hri(bob))\}$ represents the fact that

the query *on_table*(*bobs_mug*)? should be answered by perception and the query *white_coffee(bob)*? should be submitted to Bob. Formally a knowledge acquisition scheme is defined as follows:

Definition 1 (Knowledge Acquisition Scheme)**.** Let *st* be a statement that is possibly-derivable with respect to D_M and the set of open-ended literals $L_x = \bigcup_{1 \leq i \leq n} \{l_i\}$. Moreover let *KS* be the set of knowledge sources. A set $kas := \{\bigcup_{1 \leq i \leq n} \{(l_i, k_i)\} | k_i \in \mathcal{KS}\}$ is called a *knowledge acquisition scheme* for *st* w.r.t. D_M. If $L_X = \emptyset$, then the corresponding knowledge acquisition scheme is also $\emptyset$.

However, 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 method instance 2 and 3 for the situation illustrated by Figure 1. 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 2 (Possibly-acquirable)**.** An acquisition (l, ks) is called *possibly-acquirable* w.r.t. to a domain Model D_M iff there is an applicable or possibly-applicable planning step for the knowledge acquisition task $det(l, I, C, ks)$ such that I are all derivable instances of l w.r.t. D_M and C is the context. Moreover, a knowledge acquisition scheme *kas* is called possibly-acquirable iff all $(l, ks) \in kas$ are possibly-acquirable.

Let $\mathcal{D}$ be the set of domain models, $\mathcal{TL}$ be the set of task lists, $\mathcal{P}$ be the set of plans and $\mathcal{KAS}$ be the set of knowledge acquisition schemes. We call $ps \in \mathcal{D} \times \mathcal{TL} \times \mathcal{P} \times \mathcal{KAS}$ a *planning state*. A planning state is called *final* if the task list is empty and called *intermediate* if the task list is not empty. ps_D denotes the domain model, ps_t the task list, ps_p the plan and ps_{kas} the knowledge acquisition scheme of a planning state *ps*.

The term *planning step* is used in this work as an abstraction of (HTN) methods and planning operators. A planning step $\mathfrak{s}$ is represented by a 4-tuple $(\mathfrak{s}_{task}, \mathfrak{s}_{cond}, \mathfrak{s}_{eff}, \mathfrak{s}_{cost})$. $\mathfrak{s}_{task}$ is an atomic formula that describes for which task $\mathfrak{s}$ is relevant, $\mathfrak{s}_{cond}$ is a statement that constitutes the precondition of $\mathfrak{s}$, $\mathfrak{s}_{eff}$ is the effect of the $\mathfrak{s}$, and $\mathfrak{s}_{cost}$ represents the expected cost of the plan that results from the application of $\mathfrak{s}$.

Let *PS* be the set of planning states. $\mathfrak{s}_{eff}$ is a function $\mathfrak{s}_{eff} : PS \rightarrow PS$. Thus, 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 list, adding a ground instance of this task to the plan and updating the domain model according to the effects of the operator. In contrast, HTN methods transform the current planning state by replacing an active task by a number of subtasks.

Furthermore, we define the concept of a *possibly-applicable* planning step introduced in Section 2.1 as follows:

Definition 3 (Possibly-applicable)**.** A planning step $\mathfrak{s}$ is called *possibly-applicable* w.r.t. a domain model D_M and a knowledge acquisition scheme *kas* iff *kas* is possibly-acquirable and a knowledge acquisition scheme for $\mathfrak{s}_{cond}$.

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 1. 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 consider by the expected cost. 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.

For example, 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 `det(open(door2),∅,∅,percept)` is 50 (see Figure 2) and the cost of performing the task `det(connect(lab,X,kitchen), [connect(lab,door1,kitchen), connect(lab, door2,kitchen)],open(X),percept)` is 300. In this situation the cost of method instance 1 is 100, the cost of method instance 2 is $100 + 50 = 150$, and the cost of method instance 3 is $100 + 50 + 300 = 450$. Thus, in this case the applicable instance has the less expected cost. However, this does not always have to be the case.

2.3.2 Algorithm

The simplified algorithm of the proposed HTN planning system is shown by Algorithm 1. The algorithm is an extension of the SHOP (Nau et al., 1999) algorithm that additionally considers possibly-applicable decompositions.

A planning state is the input of the recursive planning algorithm. If the task list of the given planning state is empty, then the planning process successfully

Algorithm 1: Plan(ps).

```
   Result: a planning state ps', or failure
1  if ps is a final planning state then
2      return ps;
3  steps ← {(𝔰, σ, kas) | 𝔰 is the instance of a
     planning step, σ is a substitution such that 𝔰σ is
     relevant for the next task, 𝔰 is applicable or
     possibly-applicable w.r.t. ps_D and the
     knowledge acquisition scheme kas};
4  if choose (𝔰, σ, kas) ∈ steps with the minimum
     overall cost then
5      if kas = ∅ then
6          ps' ← 𝔰_eff(ps);
7          ps'' ← plan(ps');
8          if ps'' ≠ failure then
9              return ps'';
10     else
11         return (ps_D, ps_t, ps_p, kas);
12 else
13     return failure;
```

generated a complete plan and the given planning state is returned. Otherwise, the algorithm successively chooses the applicable or possibly-applicable step with the lowest expected cost. 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 it applies the step and recursively calls the planning algorithm with the updated planning state (line 5-9).

In contrast, if the planner chooses an only possibly-applicable planning step, then it stops the planning process and returns the current (intermediate) planning state including the knowledge acquisition scheme of the chosen planning step (line 10-11). In this way the planner automatically decides whether it is more reasonable to continue the planning or to first acquire additional information. In other words, it 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.

3 CONTINUAL PLANNING AND ACTING

The overall idea of the proposed continual planning and acting system is to interleave planning and acting so that missing information can be acquired by means of active information gathering. In Section 2 we described a new HTN planning system for open-ended domains. Based on that, we describe the high-level control system ACogControl in this section.

The overall architecture is sketched in Figure 3. The central component in this architecture is the *controller*. When the agent is instructed to perform a list of tasks then this list is sent to the controller. The controller calls the planner described in Section 2 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. 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 memory system. The executor stores information about the executed actions 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 acquisition attempts. Knowledge acquisition attempts are stored to avoid submitting the same query more than once to a certain knowledge source.

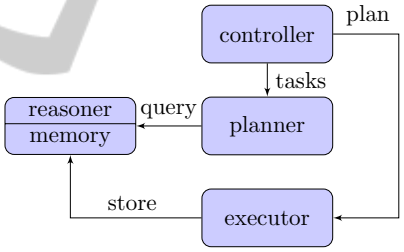

Figure 3: Illustration of the planning-based control architecture.

The behavior of the controller is specified by Algorithm 2. When the controller is invoked it first constructs an initial planning state based on the given task list and invokes the planner (lines 1-2). 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.

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, chooses the knowledge acquisition with the minimum expected cost, performs the knowledge acquisition task and continues to perform the remaining tasks. Please note that knowledge acquisition tasks can also require it to perform additional knowledge acquisition tasks. Which tasks still need to be performed in order to perform the initial

Algorithm 2: Perform($tasks$).

```
1  ps ←create-intial-ps(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 ←perform(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 |  |  perform([t_ac]);
12 |  tasks_rem ← memory.remaining-tasks();
13 |  perform(tasks_rem);
```

task list (i.e., the remaining tasks) can easily be deduced by the memory, since the memory retains knowledge of all actions that have already been executed. It is more difficult to determine which part of the already generated plan should be executed. 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. However, due to the fact this is not always the best strategy it is possible to specify domain specific control rules.

4 EXPERIMENTAL RESULTS

In this section, we present a simple case study with a mobile robot and a set of simulated experiments with several domains.

4.1 A Case Study with a Mobile Robot

The proposed planning based control system is implemented on a mobile service robot platform TASER. We performed a first simple test case in the office environment of our institute in order to demonstrate the system behaviour. The only used external knowledge source in this test case is perception. The robot was instructed to perform the task of delivering a mug (Bob's mug) into the kitchen. In this test run the robot has no information about the state of doors and therefore cannot generate a complete plan in advance.

The robot successfully performed the task. The overall execution is composed of six planning and execution phases as illustrated in Figure 4. Actions that are directly executed by a corresponding robot control program are printed blue and marked with the symbol "▶". All other tasks are non-primitive and cannot be directly executed. The fact that only a partial plan exists for a task is illustrated by a subsequent "[...]". Furthermore, the result of a sensing action is shown under the corresponding task.

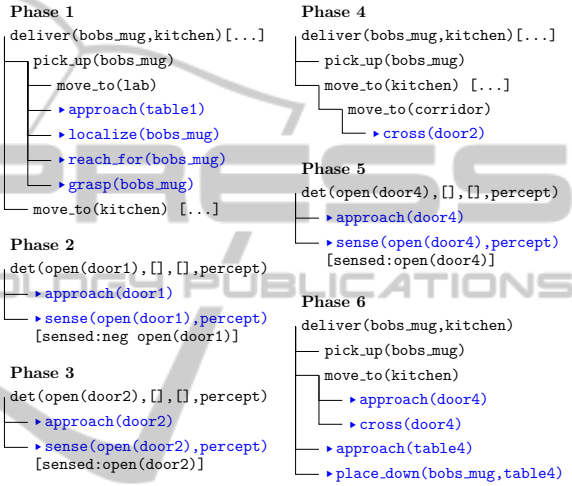

Figure 4: Execution phases of the full system test case.

At the first planning phase the planner generates a complete plan that determines how to pick up Bob's mug. Non-primitive tasks that have no subsequent "[...]" and are not further decomposed usually indicate the situation that nothing has to be done to perform the task. For example, in the first phase the task $move\_to(lab)$ is not further decomposed, because the robot initially is in the lab. Due to the fact that the planner had no information about the state of the doors it could not generate a plan for the task $move\_to(kitchen)$. The planner decides to execute the plan for $pick\_up(bobs\_mug)$ and then starts the second planning and execution phase in order to determine whether the first lab door is open. During the second execution phase the robot determines that the first lab door is closed. In order to avoid the more expensive door opening procedure the planner decides to determine whether the second lab door is open at the third planning and execution phase. The robot determines that the second lab door is open and can continue to perform the initial task (i.e., bring Bob's mug into the kitchen). In the fifth phase, the robot determines that the kitchen door is open. After the fifth phase all necessary information is available and the

robot successfully finishes its task in the last execution phase.

4.2 ACogSim

Providing an environment for the evaluation of continual planning is not a trivial task (Brenner and Nebel, 2009). We implemented a simulator, namely *ACogSim*, for the environment in order to make it possible to systematically evaluate the whole high-level control architecture—including execution—described in Section 3. The ACogSim simulator works similar to MAPSIM as described in (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 precondition 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 the ACogSim instance. The result of a sensing action $sense(l, I, C, ks)$ 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.

4.3 Performing Tasks with a Decreasing Amount of Initial Knowledge

We used ACogSim in order to evaluate the behavior of the overall control system for several domains. The objective of the conducted experiments is to determine the behavior of the system in situations where an agent needs additional information to perform a given task, but sufficient information can in principle be acquired by the agent.

4.3.1 Setup

We used an adapted version of the rover domain with 1756 facts and an instance of the depots domain with 880 facts from IPC planning competition 2002; an instance of an adapted blocks world domain with 2050 facts; and a restaurant (109 facts) and an office domain (88 facts) used to control a mobile service robot.

All domain model instances contain sufficient information to generate a complete plan without the need to acquire additional information. The simulator (ACogSim) is equipped with a complete domain model. In contrast, the agent has only an incomplete domain model where a set of facts has randomly been removed. For each domain the agent always had to perform the same task.

The objective of this experimental setup is to get deeper insights into the performance of the proposed control system. In particular, we are interested in finding an answer to the following questions: Is ACogControl always able to perform the given task? How often switches ACogControl between planning and acting? How much time is necessary for the whole planning and reasoning process? How long is an average planning phase? How does the performance change with a decreasing amount of initial knowledge?

We conducted 10 experiments for all domains with 1000 runs per experiment, except for the last experiment where 1 run was sufficient. Let f_{all} be the number of facts in a domain, then $\frac{i}{10} f_{all}$ facts were removed in all runs of the ith experiment from the domain model of the agent. Hence, in the last experiment all facts are removed (for each domain) from the agent's domain model.

The experiments where conducted on a 64-bit Intel Core 2 Quad Q9400 with 4 GB memory.

4.3.2 Results

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 average number of necessary planning and execution phases is show in Figure 5. The average number of planning and execution phases increases with a decreasing number of initial information, since the agent needs to stop the planning process and execute knowledge acquisition activities more often. We also expected the overall CPU time of the reasoning and planning process to increase for all domains with a decreasing amount of initial knowledge. However, Figure 6 shows that this is only true for the rover, the office and the restaurant domain. The blocks and the depots domain show a different behavior. For these domains the overall CPU time increases until 60 respectively 80 percent of the facts are removed from the domain model of the agent and then decreases until all facts are removed. The results shown in Figure 7 might give an explanation for this. They show that the average time for a planning phase decreases with a decreasing amount of information that initially is available for the agent. Together with the results shown in Figure 5 these results indicate that the more planning phases are performed the shorter are the individual phases. Thus, the proposed continual planning system, so to speak, partitions the

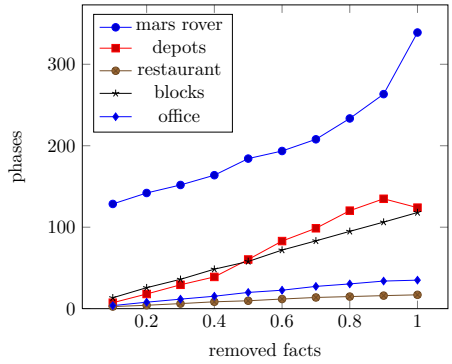

Figure 5: Average number of planning and execution phases.

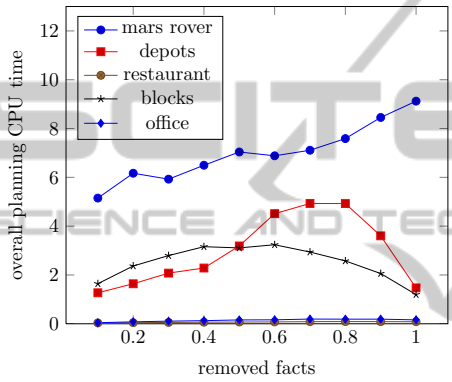

Figure 6: Average CPU time of the overall planning and reasoning process.

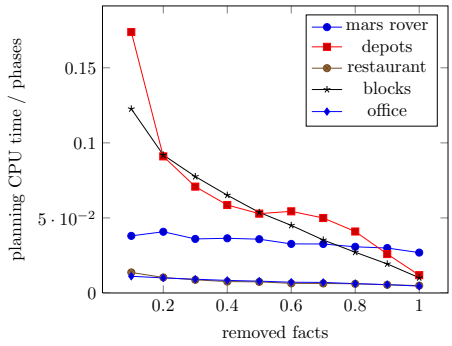

Figure 7: Average CPU time of a single planning phase.

overall planning problem into a set of simpler planning problems. Moreover, the depots and the blocks world domain indicate that the sum of the individual planning phases can be lower even if the number of planning phases is higher as shown by Figure 6.

5 RELATED WORK

Most of the previous approaches 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). Several planning approaches that generate conditional plans, including (Ambros-Ingerson and Steel, 1988; Etzioni et al., 1992; Golden, 1998; Knoblock, 1995), use *runtime variables* for the purpose of representing unknown information. Runtime variables can be used as action parameters and enable the reasoning about unknown future knowledge. Nevertheless, the information represented by runtime variables is limited since the only thing that is known about them is the fact that they have been sensed. Furthermore, planning approaches that generate conditional plans are computationally hard, scale badly in open-ended domains and are only applicable if it is possible to foresee all possible outcomes of a sensing action (Ghallab et al., 2004; Brenner and Nebel, 2009).

The most closely related previous work is (Brenner and Nebel, 2009). The proposed continual planning system also deals with the challenge of generating a plan without initially having sufficient information. 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 Golog family of action languages—which are based on the situation calculus (Reiter, 2001)—have received much attention in the cognitive robotics community. The problem of performing tasks in open-ended domains is most extensively considered by the IndiGolog language (Giacomo and Levesque, 1999), since programs are executed in an on-line manner and thus the language to some degree is applicable to situations where the agent posses only incomplete information about the state of the world. Regrettably, IndiGolog only supports binary sensing actions.

Besides Golog the only other known agent programming language is FLUX (Thielscher, 2005) which is based on the Fluent Calculus. FLUX is a powerful formalism, but uses a restricted form of con-

ditional planning. As already pointed out, conditional planning is not seen as an adequate approach for the scenarios we are interested in.

6 DISCUSSION AND CONCLUSIONS

State-of-the-art planning techniques can provide artificial agents to a certain degree with autonomy and robustness. Unfortunately, reasoning about external information and the acquisition of relevant knowledge has not been sufficiently considered in existing planning approaches and is seen as an important direction of further growth (Nau, 2007).

We have proposed a new continual HTN planning based control system that can reason about possible, relevant and possibly-acquirable extensions of a domain model. It makes an agent capable of autonomously generating and answering relevant questions. The domain specific information encoded in HTN methods not only helps to prune the search space for classical planning problems but can also nicely be exploited to rule out irrelevant extensions of a domain model.

Planning in open-ended domains is obviously more difficult than planning based on the assumption that all information is available at planning time. Nevertheless, the experimental results indicate that the proposed approach partitions the overall planning problem into a number of simpler planning problems. This effect can make continual planning in open-ended domains sufficiently fast for real world domains. Additionally, it should be considered that the execution of a single action is often much more time intensive for several agents (e.g., robots) than the planning phases of the evaluated domains.

Like classical HTN planning the proposed continual planning and acting based control system is *domain-configurable*[4]. This means that the core planning, reasoning and controlling engines are domain independent, but can exploit domain specific information. For all evaluated domains we only defined a few simple HTN methods. We expect that the evaluation results will be significantly better if one adds more sophisticated domain specific information to the domain models.

ACKNOWLEDGEMENTS

This work is founded by the DFG German Research Foundation (grant #1247) – International Research Training Group CINACS (Cross-modal Interactions in Natural and Artificial Cognitive Systems).

[4]as described in (Nau, 2007)

REFERENCES

Ambros-Ingerson, J. A. and Steel, S. (1988). Integrating planning, execution and monitoring. In *AAAI*, pages 83–88.

Brenner, M. and Nebel, B. (2009). Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331.

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.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning Theory and Practice*. Elsevier Science.

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.

Golden, K. (1998). Leap before you look: Information gathering in the puccini planner. In *AIPS*, pages 70–77.

Knoblock, C. A. (1995). Planning, executing, sensing, and replanning for information gathering. In *IJCAI*, pages 1686–1693.

Littman, M. L., Goldsmith, J., and Mundhenk, M. (1998). The computational complexity of probabilistic planning. *J. Artif. Intell. Res. (JAIR)*, 9:1–36.

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.

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.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Thielscher, M. (2005). FLUX: A logic programming method for reasoning agents. *Theory Pract. Log. Program.*, 5:533–565.