Open-Ended Domain Model for Continual Forward Search HTN Planning

Dominik Off and **Jianwei Zhang**
TAMS, Department of Informatics, University of Hamburg
Vogt-Kölln-Strasse 30, 22527 Hamburg, Germany
{off,zhang}@informatik.uni-hamburg.de

Abstract

Domain models for automated planning systems often rely on the closed world assumption. Unfortunately, the closed world assumption is unreasonable in many real world planning domains. We propose an open-ended domain model based on definite clauses that can be flexibly extended and is able to automatically determine relevant but unknown information. The determination of relevant but unknown information is intended to be the starting point of an active information gathering process which might result in the enablement of additional planning alternatives. This is particularly relevant for situations in which it would otherwise be impossible to find any plan at all. Moreover, we present several knowledge representation constructs that help to deal with the special challenges of open-ended domains. The proposed domain model is mainly intended for hierarchical task network planning systems that generate plans in open-ended domains by means of interleaving planning and knowledge acquisition.

Introduction

Planning systems have been developed that in principle are efficient enough to solve realistic 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) nicely summarized this problem as follows:

> 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.

We propose a domain model that is able to answer some of the questions raised in the above quotation. More precisely, the main contributions of this work are:

1. We propose an open-ended domain model—called *Domain Model for Artificial Cognitive Systems (ACogDM)*—based on definite clauses that is able to answer the questions: 'What to look for?' and 'Where to get it?'.
2. We demonstrate how the proposed language of the domain model can be easily extended by additional constructs.
3. We propose extensions of our basic domain model that help to deal with the special requirements (e.g., computational complexity) for open-ended domains.

The proposed domain model is particularly intended for forward search (i.e., forward decomposition) *Hierarchical Task Network (HTN)* (Ghallab, Nau, and Traverso 2004) planning approaches. However, it might also be useful for other approaches.

A domain model for planning is usually composed of information about the state of the domain and information about the possible activities of an agent. We call the former part of a domain model the *state model* and the later part the *activities model*.

Extendable State Model

Several non-classical planning systems use axiomatic inference techniques to reason about the state of the world (Ghallab, Nau, and Traverso 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, Hoffmann, and Nebel 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). We agree with this argumentation, but only under the assumptions that (1) a complete state model should be represented and (2) 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 which—as introduced later—permits the explicit representation of complete parts at the level of predicates. Thus, we believe that it is reasonable to directly represent negative information in the context of this work.

We use *definite clauses* 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). In short, a definite clause is notated as $A_0 \leftarrow A_1, \ldots, A_n$ whereas $n \geqslant 0$. 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, we use '$\not\vdash$' in definite clauses and definite goals as the *negation as (final) failure* operator as introduced by (Clark 1987) and implemented in several prolog systems.

Additionally, we introduce two special kinds of atomic formulas: *literals* and *statements*. If f is an atomic formula, then we call f and $\texttt{neg}\ f$ a literal. Furthermore, we call st 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

Literals and statements are syntactically defined as atomic formulas for the purpose of reasoning about them in the language of definite clauses. 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—including literals—are always implicitly quantified. Statements in a definite clause are (implicitly) universally quantified and statements that constitute a definite goal are (implicitly) existentially quantified.

Similar to PDDL with PDDL Axioms our 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 we additionally support the explicit representation of negative information. Moreover, we support 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 *concepts*. In principle, our state model can be extended to support any conceptual knowledge as long as we can compile this information to the underlying knowledge representation formalism, namely, a set of definite clauses. We exploit this feature in the following part of the paper by successively adding support for additional concepts that are intended to deal with the special requirements of open-ended domains. For example, we are going to support the explicit representation of subsumption-relations (e.g., 'A mug is an object').

A *state model* is formally defined as follows:

Definition 1 (state model)**.** A *state model* is a quadruple $s_M = (F, C, R_D, R_G)$. F is a set of literals and C is a set of atomic formulas such that $F \cap C = \emptyset$. R_D is a set of definite clauses $l \leftarrow s$ 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, 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.

The fact that a state model constitutes a definite program has the advantage that the semantics of a state model is based on the well-known semantics of a definite program. From a more practical perspective, we can additionally benefit from the actuality that several highly optimized prolog implementations are available that can automatically determine whether a definite goal can be proved or not.

Based on the semantics of a definite program we can define the derivability of an atomic formula as follows:

Definition 2 (derivable)**.** An atomic formula f is *derivable* with respect to a state model s_M and a grounding substitution σ (denoted as $s_M \vdash_\sigma f$) iff $f\sigma$ is a logical consequence[1] of ${s_M}^{dp}$.

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

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

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

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

$$\texttt{neg neg}\ st \leftarrow st \tag{GR4}$$

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

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

These rules determine the semantics of statements. In particular, the handling of the introduced negation operator '$\texttt{neg}$' is specified. Furthermore, please note that it directly

[1] More precisely, this means that f is a member of the least Herbrand model $M_{{s_M}^{dp}}$ (Nilsson and Maluszynski 1995, Theorem 2.16).

follows from Definition 2 that a statement st is derivable with respect to a state model s_M and a substitution σ iff $st\sigma$ is a logical consequence of ${s_M}^{dp}$. For the purpose of avoiding misunderstandings we would like to emphasize again that statements are syntactically treated as atomic formulas, but semantically constitute a first order logic sentence.

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, Hoffmann, and Nebel 2005). Two example axioms are defined as follows:

$$in\_room(O, R) \leftarrow on(O, T), in\_room(T, R) \quad \text{(DR1)}$$

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

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.

Due to the fact that we support the explicit representation of negative information it is in principle possible to construct a *syntactically inconsistent*[2] state model. This means that it is possible to construct a state model where st and neg st are derivable. Indeed, it is desirable to support domain modellers with software tools in order to prevent the creation of inconsistent state models. 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 $closed(door1)$ is derivable. Thus, one also has to deal with inconsistency in CWA based models. Furthermore, note that the explicit representation of negation by means of the definite clause neg $open(door1) \leftarrow closed(door1)$ has the advantage that it would make it possible to detect the semantic inconsistency via syntactic techniques. However, we will not further address that problem here, since dealing with consistency is not in the focus of this work. The fact that a state model s_M is consistent is denoted as $\mathfrak{c}(s_M)$. In the following part of this paper it is implicitly always assumed that a state model is consistent.

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.

[2]See (Nguyen 2008) for more details about syntactic and semantic inconsistencies.

Example 1 For example, let us assume that the only literals that can be derived from the state model s_M of an agent are $mug(bobs\_mug)$ and $in\_room(bobs\_mug, kitchen)$. If one would try to derive whether a true instance of $mug(X) \wedge color(X, red)$ 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 exist. 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 statement $in\_room(bobs\_mug, \mathit{office})$. 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 domain 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. Moreover, in the latter case the domain model should make it possible to derive all situations in which the existence of an additional instance is possible. If we want to enable such a reasoning, then we need an open-ended domain model. We propose an open-ended domain model that is based on the following three concepts: a *F-extension*; an *open-ended literal*; and a *possibly-derivable statement*.

For the purpose of reasoning about open-ended domains we have to reason about possible extensions of a state model. Here we only consider extensions 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 (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 world model stays consistent. We denote the set of all instances of a statement st that are derivable with respect to a state model s_M as $\widetilde{\vdash}(s_M, st)$ respectively as $\widetilde{\vdash}(st)$ if the respective state model is apparent. Furthermore, we call literals for which the existence of non-derivable instances is possible *open-ended*:

Definition 4 (open-ended literal)**.** A literal l is called *open-ended* w.r.t. a state model s_M (denoted as $l^{\sqsubseteq}$) iff it is possible that there is an instance $l\sigma$ of l and a state model s'_M such that $s_M \sqsubseteq_F s'_M$, $l\sigma \notin \widetilde{\vdash}(s_M, l)$ and $l\sigma \in \widetilde{\vdash}(s'_M, l)$.

Please note that for a ground literal the following holds:

Remark 1. If l is ground, then l is open-ended iff $\nvdash l$ and $\nvdash$ neg l holds.

Let us recall the situation of Example 1 in order to exemplify the concept of an open-ended literal. $mug(X)$ and $color(red, X)$ 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 $non\text{-}ground(l)$ 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:

$$l^{\sqsubseteq} \leftarrow non\text{-}ground(l) \quad \text{(GR7)}$$

$$l^{\sqsubseteq} \leftarrow ground(l), \not\vdash l, \not\vdash (\texttt{neg}\ l) \quad \text{(GR8)}$$

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).

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

Definition 5 (possibly-derivable statement)**.** A statement st is possibly-derivable w.r.t. to a state model s_M and a set of open-ended literals L_x (denoted as $\Diamond(st, L_x)$) iff the existence of a new instance $l\sigma$ for each $l \in L_x$ implies the existence of a new instance $st\sigma$ of st.

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 w.r.t. 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 1 and would like to know whether the statement $st = mug(X) \wedge color(X, red)$ is possibly-derivable (i.e., 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 $\{color(bobs\_mug, red)\}$. In the second situation, we exploit the fact that there might exist an unknown red mug and st is possibly-derivable with respect to s_M and the resulting set of open-ended literals $\{mug(X), color(X, red)\}$.

Let $literal(l)$ be a meta-predicate that holds iff l is a literal. In order to be able to derive possibly-derivable statements we introduce the following generic rules:

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

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

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

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

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

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

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

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

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 w.r.t. this extension. Note that a (non-ground) literal can be both derivable and open-ended. L_x denotes the set of open-ended literals of the previous part of a statement and initially is empty (see GR9). Including L_x 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 possible 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, GR13 - GR16 essentially describe well-known rules of first order logic.

Continual Planning in Open-Ended Domains

As already mentioned, ACogDM is developed for forward decomposition HTN planners (e.g., SHOP (Nau et al. 1999) or SHOP2 (Nau et al. 2003)). In this section we briefly motivate and explain how the proposed domain model can be combined with such a planner so that the combination constitutes a continual planning system.

Forward decomposition HTN planners choose between a set of *relevant* (Ghallab, Nau, and Traverso 2004) methods or planning operators (i.e., actions) that can be in principle applied to the current *task network*. In the context of this work preconditions of action or methods are represented by definite goals of the form '$\leftarrow st$' such that st is a statement (e.g., $\leftarrow (mug(X) \wedge \texttt{neg}\ in\_room(X, kitchen))$). If it does not lead to ambiguity, then we will omit the leading '$\leftarrow$' of a definite goal. A relevant method or planning operator can actually be applied if and only if its precondition p holds (i.e., an instance $p\sigma$ 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 network) to be the set of all preconditions of relevant methods or planning operators. A HTN planner cannot 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.

The general idea is to adapt a forward decomposition HTN planner such that the behaviour is not changed as long as sufficient information is available in order to generate a plan. However, if necessary information is missing, then the planning process is stopped and a partial plan prefix and a set of open-ended literals of a relevant and possibly-derivable precondition is returned. If the planner stops the planning process due to a lack of knowledge, then the set of open-ended literals constitute a relevant extension of the domain model that would make it possible to continue the planning process. Hence, a planner can answer the question "What to look for?" as follows: Look for non-derivable instances of the open-ended part (i.e., a set of open-ended literals) of possibly-derivable and relevant preconditions. For example, if want a planner to perform the task "Deliver Bob's mug into the kitchen", but the the fact whether the kitchen door is open or closed cannot be derived from the domain model, then a planner returns a partial plan (e.g., $[pick\_up(bobs\_mug), \ldots]$) and a set of open-ended literals (e.g., $\{open(kitchen\_door)\}$). Based on that, a planner can try to generate and execute a plan that acquires a non-derivable instance for each open-ended literal (e.g., try to acquire whether the kitchen door is open). Subsequently, a planner can continue the planning process based on the updated domain model. By this means a planner can automatically switch between planning and acting such that missing information can be acquired by means of active information gathering.

Additional State Model Constructs

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.

In this section, we are going to extend the state model by additional concepts that 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 such that the number of open-ended literals can be reduced. For example, according to Remark 1 one could deduce that an opened-ended and ground literal l is not open-ended if additional domain knowledge would make it possible to derive l or `neg` l.

With the current state model (i.e., the state model constituted by GR1 - GR16) every non-ground literal is open-ended (see GR7). To put it another way, we assume that we never know all instances of a non-ground literal. However, this might not always be the case. On the conceptual—or semantical—level domain constraints can limit the number of possible instances of a statement. For example, let us assume that the literal $in\_room(bobs\_mug, \mathit{office})$ 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 we extend the language of the state model by constructs of the form $i_{max}(l, n, c)$ such that l is a literal, $n \in \mathbb{N} \cup \{\infty\}$ and c is a statement. $i_{max}(l, n, c)$ specifies that the literal l can maximally have n ground instances if c holds. In order to "ground" this additional construct to the level of definite clauses we have to add the following rules:

$$i_{max}(l, n) \leftarrow i_{max}(l, n, c), c \quad \text{(GR17)}$$

$$i_{max}(l, \infty) \leftarrow non\text{-}ground(l), \not\vdash i_{max}(l, n, X_1) \quad \text{(GR18)}$$

$$i_{max}(l, 1) \leftarrow ground(l) \quad \text{(GR19)}$$

Now we can formulate an advanced version of (GR7) as follows:

$$l^{\sqsubseteq} \leftarrow non\text{-}ground(l), i\_max(l, n), n < |\widetilde{\vdash}(l)| \quad \text{(GR20)}$$

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

A less flexible, but easier way to define the maximum number of instances for a subset of non-ground literals is based on the *instantiation scheme* of a literal.

A literal or a term is called *duplicate-variable-free* iff it does not contain two identical variables. For example, $p(X, Y)$ is duplicate-variable-free and $p(X, X)$ is not duplicate-variable-free. For duplicate-variable-free terms and literals we define a corresponding *instantiation scheme* as follows:

Definition 6 (instantiation scheme)**.** Let g be a duplicate-variable-free term or literal. The *instantiation scheme* g^{ρ} of g is defined as follows:

$$g^{\rho} := \begin{cases} ground & \text{if } g \text{ is ground} \\ var & \text{if } g \text{ is a variable} \\ f(u_1^{\rho}, \ldots, u_m^{\rho}) & \text{else if } g = f(u_1, \ldots, u_m) \end{cases}$$

An instantiation scheme abstracts from the concrete arguments of a literal by replacing variables with the constant var and ground terms with the constant $ground$. We restrict instantiation schemes here to duplicate-variable-free terms and literals, because the multiple occurrence of the same variable imposes additional constraints that otherwise would be unintentionally abstracted away. Moreover, from the knowledge engineering perspective we wanted to keep the definition of an instantiation scheme simple, since instantiation schemes are intended to be specified by a human domain expert. Explicitly representing possible constraints that result from duplicate variables in a literal would make the representation significantly more difficult while only being necessary for the minority of literals. Additionally, please note that Definition 6 can also be applied to negative literals, since the negation operator is technically a "normal" predicate. Let $\mathcal{L}^{\rho} := \{l^{\rho} | l \in \mathcal{L}\}$. The maximum number of possible instances with respect to an instantiation scheme is defined by the function $i_{max\_\rho} : \mathcal{L}^{\rho} \rightarrow \mathbb{N} \cup \infty$. In

ACogDM, we can define the possible number of instances with respect to a representation scheme with atomic formulas of the form $i\_max\_\rho(scheme, n)$ such that $scheme$ is a instantiation scheme and $n \in \mathbb{N} \cup \{\infty\}$. In order to support these constructs we add the following generic rule to the state model:

$$i_{max}(l, n) \leftarrow i_{max\_\rho}(l^\rho, n) \qquad \text{(GR21)}$$

For example, the fact that an object can only be in one room at a given point in time can be easily represented by the atomic formula $i\_max\_\rho(in\_room(ground, var), 1)$. 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, $i\_max\_\rho(in\_room(ground, var), 1)$ solely makes it possible to deduce that all statements with the instantiation scheme $in\_room(ground, var)$ can only have one instance.

We can omit redundancies introduced by i_max_ρ via adding generic rules. For the purpose of achieving this we first introduce the *sub-scheme*-relation as follows:

Definition 7 (sub-scheme)**.** An instantiation scheme s is called a *sub-scheme* of an instantiation scheme s' (denoted as $s \preceq s'$) iff one of the following holds:

- $s' = var$;
- $s = ground \wedge (s' = ground \vee s' = var)$;
- or $s = g(\alpha_1, \ldots, \alpha_n)$ and $s' = g(\beta_1, \ldots, \beta_n)$ and for all $1 \leqslant i \leqslant n$ it holds that $\alpha_i \preceq \beta_i$.

The *sub-scheme*-relation constitutes an ordering on instantiation schemes. We are interested in this ordering, since it is related to i_{max_ρ} as stated by the following proposition:

Proposition 1. *If l and l' are duplicate-variable-free literals, then the following holds: $l^\rho \preceq l'^\rho \Rightarrow i_{max\_\rho}(l) \leqslant i_{max\_\rho}(l')$.*

We define the *lift* of a duplicate-variable-free literal or term with respect to a compatible instantiation scheme as follows:

Definition 8 (lift)**.** Let g be a duplicate-variable-free literal or term, g'^ρ be an instantiation scheme such that $g^\rho \preceq g'^\rho$ and $X^\star$ denote a new (i.e., unused) variable. The *lift* of g w.r.t. g'^ρ is defined by the function $\rho_\uparrow$ as follows:

- $\rho_\uparrow(g, g'^\rho) := g$; if $g^\rho = g'^\rho$
- $\rho_\uparrow(g, g'^\rho) := X^\star$; if $g'^\rho = var$ and g is not a variable
- $\rho_\uparrow(g, g'^\rho) := f(\rho_\uparrow(u_1, {u'_1}^\rho), \ldots, \rho_\uparrow(u_m, {u'_m}^\rho))$; if $g = f(u_1, \ldots, u_m)$ and $g'^\rho = f({u'_1}^\rho, \ldots, {u'_m}^\rho)$

Lifting a literal or a term with respect to an instantiation scheme g^ρ essentially means to replace ground terms by new variables such that the instantiation scheme of the resulting literal is g^ρ. For example, lifting $in\_room(bobs\_mug, office)$ with respect to the instantiation scheme $in\_room(ground, var)$ results in $in\_room(bobs\_mug, X)$. Now we can propose the following:

Proposition 2. *For each duplicate-variable-free literal l,* neg *l is derivable w.r.t. a state model s_M if the following holds:*

1. $\neg\exists_\sigma : s_M \vdash_\sigma l$; *($l$ is not derivable)*
2. $\exists_{l'^\rho \in \mathcal{L}^\rho} : \; l^\rho \preceq l'^\rho \wedge |\widetilde{\vdash}(s_M, \rho_\uparrow(l, l'^\rho))| = i_{max\_\rho}(l'^\rho)$

This means that we can derive neg l if l is not derivable and it exists an instantiation scheme that is more general than the instantiation scheme of l for which all possible instances are already derivable. Proposition 2 constitutes a rule that enables us to now derive, based on the definition of i_{max_ρ}, that something cannot hold. We can now represent Proposition 2 as the following rule:

$$\begin{aligned} \texttt{neg}\ l \leftarrow {} & \nvdash l, i_{max_\rho}(l'^\rho, n), l^\rho \preceq l'^\rho, \\ & |\widetilde{\vdash}(\rho_\uparrow(l, l'^\rho))| = n \end{aligned} \qquad \text{(GR22)}$$

For example, we can now derive 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.

We proposed an open-ended state model where all statements are by default interpreted based on the *open world assumption (OWA)*. 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. This property of a predicate is called the *interpretation model* of a predicate and can either be OWA or CWA. 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 situation it would be reasonable to define the interpretation model of the `connection` predicate as CWA. This definition implies neg `connection(R1,D,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 CWAs reduces the lack of knowledge and can significantly improve the performance of the plan generation and knowledge acquisition process.

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^ϱ. The fact that a predicate is interpreted with respect to the CWA is represented by atomic formulas 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. In order to support CWAs at the level of predicates we only have to add the following rule:

$$\texttt{neg}\ l \leftarrow cwa(l^\varrho), \nvdash l \qquad \text{(GR23)}$$

Another featured knowledge representation construct is the explicit definition of *subsumption*-relations between literals. More precisely, subsumption is a relation between

concepts which are constituted by literals. The subsumption relation 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 literal $p(X_1, \ldots, X_n)$ is conceptually subsumed by a literal $p'(X_1, \ldots, X_n)$ is denoted as $p(X_1, \ldots, X_n) \sqsubseteq p'(X_1, \ldots, X_n)$. Information about subsumption relations can now be exploited as follows:

$$l \leftarrow l' \sqsubseteq l, l' \tag{GR24}$$

In other words, a literal is derivable if there is a subconcept that is derivable. Moreover, it can be easily shown that the following holds:

$$\texttt{neg}\ l' \sqsubseteq \texttt{neg}\ l \leftarrow l \sqsubseteq l' \tag{GR25}$$

Similarly, the fact that the literals $p(X_1, \ldots, X_n)$ and $p'(X_1, \ldots, X_n)$ are disjunct is denoted as $p(X_1, \ldots, X_n) \sqcap p'(X_1, \ldots, X_n)$. Knowledge about the disjointness is exploited by the following inference rule:

$$\texttt{neg}\ l \leftarrow l \sqcap l', l' \tag{GR26}$$

Knowledge Acquisition

We already proposed an answer to the question: "What to look for?". In this section we briefly survey our answer to the second initial question:"Where to get it?".

The central concept to answer this question is an *external knowledge source*. Anything that is able to provide additional information about the world (e.g., perception, human-computer interaction, low-level reasoning and planning) might serve as an external knowledge source. Of course, an external knowledge source has to conform to a corresponding interface in order to enable the planner to submit queries to various external sources in an uniform manner.

Artificial agents—especially robots—can usually acquire information from a multitude of sources. Sources may differ strongly from each other in terms of the type of information they can provide and other non-functional characteristics (e.g., acquisition cost, reliability, degree of necessary human interaction, world altering effects). Here we restrict ourselves to the following two major properties of an external knowledge source: the type of information it in principle can provide, and how expensive it is to answer a certain question. Let ks be the symbolic representation of a knowledge source and l be a literal. We further extend the representation language of our state model by constructs of the form $applicable\_ks(ks, l)$ in order to denote that ks is in principle able to provide new instances of l.

Now we can answer the question "Where to get it?" with: "You can get the desired information from an applicable knowledge source".

How expensive it is to acquire new information from external sources strongly depends on: the information one is looking for, the chosen knowledge source, and the current situation. The expense that takes these three issues into account is called the *acquisition cost*. The fact that the cost to acquire a new instance of a literal l from a knowledge source ks is c is specified by constructs of the form $ac(ks, l, c)$.

Based on the applicability of external knowledge sources and the expected acquisition cost, a planner can decide (i.e., plan) how to acquire relevant information (i.e., the open-ended part of a relevant precondition) from external knowledge sources.

Activities Model

The activities model of ACogDM contains knowledge about *planning steps* and *tasks*. The term *planning step* is used as an abstraction of *planning operators* (i.e., actions), *HTN methods* and *High-level actions (HLAs)*. Planning operators and HTN methods are mainly defined as in (Ghallab, Nau, and Traverso 2004) and HLAs are mainly defined as in (Russell and Norvig 2010). Additionally, it is possible to specify a cost for each planning step. Please note that specifying the cost of an action or an HTN method is not a new idea and, for example, also supported by SHOP2 (Nau et al. 2003).

In many domains there are tasks respectively goals for which a lot of possible solutions exist. However, in the light of additional domain knowledge one can often significantly reduce the number of possible plans and thereby reduce the computational effort. Continual planning approaches benefit to a special degree from the reduction of alternative solutions, because less alternatives usually also means less unnecessary execution of planning operators. And execution is often (e.g., in robotics) a time intensive process.

For forward-search HTN planning (e.g., SHOP (Nau et al. 1999)) we propose to support additional domain knowledge which makes it possible to reduce the number of alternative plans for a given task.

Example 2 For example, let us assume that we instruct a robot to pick up Bob's mug from the kitchen table ($pick\_up(bobs\_mug, kitchen\_table)$) and there is exactly one HTN methods that always decomposes this task into the subtasks $[goto(kitchen\_table), grasp(bobs\_mug)]$. Moreover, let us assume that there are in principle several different ways to go into the kitchen. Nevertheless, how the robot actually performs the task of going into the kitchen does not affect the task of grasping Bob's mug. This information can be exploited in a situation where a planner successfully generated a plan for the purpose of getting into the kitchen and then realizes that it is impossible to grasp Bob's mug (e.g., because the mug is in another room). In this situation it obviously does not make sense to backtrack and try to find an alternative plan for the task $goto(kitchen\_table)$. A planner that knows that these tasks can be solved independently can first generate a sufficiently good plan for $goto(kitchen)$, cut alternative decompositions for $goto(kitchen)$ and then plan to grasp Bob's mug.

In ACogDM it is possible to express the independency of subtasks by task lists of the form $[\{t_1, \ldots, t_m\}, \ldots, \{t_{m+k}, \ldots, t_{m+k+j}\}]$ such that one can plan individually for each set of tasks embraced by '{}'. For Example 2 one can represent the subtasks of $pick\_up(bobs\_mug, kitchen)$ as $[\{goto(kitchen\_table)\}, \{grasp(bobs\_mug)\}]$.

Related Work

Most existing automatic theorem proving or knowledge representation and reasoning systems, including planning domain models, 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, MacGregor, and Russ 2010) and has also been proposed by (Etzioni, Golden, and Weld 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.

Converting knowledge from one representation scheme to another in general and particularly converting an ontology (e.g., a description logic based representation) to a definite program is not a new idea. The integration of description logic and logic programming is currently an active research topic (Motik and Rosati 2007). How an OWL based ontology can be converted to prolog programs is described in (Samuel et al. 2008). Furthermore, it is possible to express a subset of OWL directly as a logic program, namely, a *description logic program* (Hitzler, Studer, and Sure 2005). Description logic has been used in many different aspects in planning systems (Gil 2005). An approach that combines HTN planning and description logic reasoning is described by (Hartanto and Hertzberg 2008).

Discussion and Conclusion

We have presented an open-ended domain model based on definite clauses that can be extended by additional constructs. The proposed conceptualization of open-endedness allows us to automatically determine relevant but unknown information which makes additional planning alternatives possible. In particular, it often makes it possible to find any plan at all if insufficient information is a priori available.

We observe definite clauses to be a solid representational basement that makes it relatively easy to extend the state model language by additional constructs. Furthermore, we define several additional state model constructs that help to deal with the special challenges of open-ended domains as well as exemplify how a basic state model can be successively extended. The additional state model constructs so to speak reduce the "open-endedness" of a state model by enabling it to rule out possible extensions of a state model.

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)

References

Chalupsky, H., and Russ, T. A. 2002. Whynot: Debugging failed queries in large knowledge bases. In *AAAI/IAAI*, 870–877.

Chalupsky, H.; MacGregor, R. M.; and Russ, T. 2010. *PowerLoom Manual (Version 1.48)*. University of Southern California, Information Sciences Institute.

Clark, K. L. 1987. Negation as failure. *Logic and databases* 293–322.

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)*, 114–121. AAAI Press.

Etzioni, O.; Golden, K.; and Weld, D. S. 1997. Sound and efficient closed-world reasoning for planning. *Artif. Intell.* 89(1-2):113–148.

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

Gil, Y. 2005. Description logics and planning. *AI Magazine* 26(2):73–84.

Hartanto, R., and Hertzberg, J. 2008. Fusing dl reasoning with htn planning. In *KI*, 62–69.

Hitzler, P.; Studer, R.; and Sure, Y. 2005. Description logic programs: A practical choice for the modelling of ontologies. In *1st Workshop on Formal Ontologies Meet Industry, FOMI'05, Verona, Italy, June 2005*.

Motik, B., and Rosati, R. 2007. A faithful integration of description logics with logic programming. In *IJCAI*, 477–482.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. Shop: Simple hierarchical ordered planner. In *IJCAI*, 968–975.

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.

Nguyen, N. T. 2008. *Advanced Methods for Inconsistent Knowledge Management*. Springer.

Nilsson, U., and Maluszynski, J. 1995. *Logic, Programming, and PROLOG*. New York, NY, USA: John Wiley & Sons, Inc.

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

Samuel, K.; Obrst, L.; Stoutenburg, S.; Fox, K.; Franklin, P.; Johnson, A.; Laskey, K. J.; Nichols, D.; Lopez, S.; and Peterson, J. 2008. Translating owl and semantic web rules into prolog: Moving toward description logic programs. In *Theory and Practice of Logic Programming*, volume 8, 301–322.

Subrahmanian, V. S. 1999. Nonmonotonic logic programming. *IEEE Trans. Knowl. Data Eng.* 11(1):143–152.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of pddl axioms. *Artificial Intelligence* 168(1-2):38–69.