PRACTICAL GOAL-BASED REASONING IN ONTOLOGY-DRIVEN APPLICATIONS

Huy Pham and Deborah Stacey
School of Computer Science, University of Guelph, Guelph, Canada

Keywords: Ontology-driven planning framework, Planning ontology, Rule-based reasoning in ontology, Ontology-driven rule-based reasoning.

Abstract: In this paper, we describe a practical and effective approach to incorporating goal-based reasoning into ontology-driven applications. We present a reusable ontology-driven planning framework that could be used for such a purpose, and provide detailed examples on how ontology-driven application designers can use this framework to describe their planning problems, translate them into equivalent rule-based programs, execute them on a planner, and get back the results. Several interesting practicality challenges are discussed, and practical solutions are also proposed.

1 INTRODUCTION

Rule-based reasoning and ontological modelling are two highly desirable features of a modern knowledge-based system. Ontology provides the system with the standardized medium needed to capture its knowledge in a domain and application independent way, while rule-based reasoning provides the system with the ability to make purposeful decisions based on this captured knowledge. Due to the clear advantages that they offers, these two complimentary features are often expected to be brought together to build intelligent and reusable knowledge-based systems. As it has been widely reported in the literature however, integrating rule-based reasoning into ontology-driven applications has been a very challenging task.

Ontology is based on Description Logics, a knowledge representation formalism designed primarily for describing and reasoning about structural knowledge, while rule-based planning is based primarily on Logic Programming, a different formalism best suited for non-monotonic reasoning. Reconciling these two formalisms is a non-trivial task for a couple of reasons. First, Description Logics use the open world assumption, in which a fact can not be assumed to be false unless it was explicitly stated so, while Logic Programming (i.e., Rules) uses the closed world assumption. Second, incorporating feature from one language into the other often renders the language undecidable, making automated reasoning in the language infeasible. Due to these inherent difficulties, existing language-reconciliation approaches to integrating planning with ontological modelling has had very limited succcess. While several ad hoc integration approaches have also been reported to have worked, a seamless and transparent framework for integrating planning capabilities into ontology-driven applications is still missing to the best of our knowledge.

In this paper, we observe that while Ontology and Description Logics do not provide built-in support for the kind of reasoning needed for rule-based planning, they are fully capable of *describing* planning problems. Based on this simple observation, we propose an intuitive ontology-driven planning framework that would allow ontology-driven application designers to seamlessly and transparently bring planning capabilities into their applications.

The paper is organized as follows. Section 2 introduces the reader to knowledge-based planning, ontological modeling, as well as existing approaches to combining knowledge-based planning with ontological modelling. Section 3 describes our proposed approach and framework, and discusses the practical advantages that it offers. Section 4 and 5 dive into the details of the proposed framework. Finally, Section 6 summarizes the discussion and discusses some future works.

Pham H. and Stacey D..
PRACTICAL GOAL-BASED REASONING IN ONTOLOGY-DRIVEN APPLICATIONS.
DOI: 10.5220/0003668300990109
In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development* (KEOD-2011), pages 99-109
ISBN: 978-989-8425-80-5

2 PLANNING IN ONTOLOGY-DRIVEN APPLICATIONS

2.1 Planning

From a computational point of view, planning is the task of coming up with a sequence of actions that will achieve a given set of goals (Russell and Norvig, 2002). An intelligent trip planner, for example, is said to be *planning* when it tries to put together a travel plan for its user. This travel plan could look something like "Take the 9:15AM bus from the Guelph's main campus to Toronto's Pearson airport, board flight AC357 to Paris's Charles de Gaulle airport, take a cab ride to the KEOD conference in Paris." Each of these travel step is called a (planning) *action*, and the sequence as a whole is called a *plan*. To find such a plan, the trip planner needs to be able to reason about several things. First, it needs to know when it can perform a particular action. Taking a bus ride, for example, is only possible if the user is at a location on the bus's route. This constraint is called a *precondition*, and can be different for each action. Also, performing a particular action is expected to produce a certain set of *effects*. Boarding a plane, for example, causes the user's location to change from the original airport to the destination airport (ignoring the flight duration). When that happens, the world is said to have changed its *state*, from one in which the user was at the original airport, to one in which he is at the destination airport. The trick, of course, is to find a plan that once executed, would result in a state (called the *goal state*) in which the user is at his or her desired destination. There might also be additional constraints regarding cost, traveling time, wait time, number of hops, etc. This task, for a software agent, is not as easy as it would be for a human. To come up with such a plan, the trip planner would have to search through a lot of possible combinations of action sequences. As the number of actions available in each step (aka, the size of the *action space*) increases, or as the length of the sequence (aka, *plan size* or *planning horizon*) increases, the amount of search the planner has to do increases combinatorically. For real-world problems, where the number of actions can be in the hundreds and plan size is in the tens, planning often becomes a prohibitive expensive process, and some techniques will need to be employed to cope with this complexity. Among the most popular of these techniques is to make use of domain heuristics and to make use of the hierarchical structure of the problem. We describe planning heuristics and hierarchical planning techniques in more details in Section 4.3.

From the example above, we can see that simple[1] planning problems can be characterized by the following types of description:

- **Actions:** What are the available actions from which a plan can be composed?
- **Actions' Pre-conditions:** Under what circumtances an action is considered possible? (These preconditions are used by the planner to avoid putting together invalid plan.)
- **Actions' Effects (aka, Post-conditions):** How each of the workflow actions, when carried out, will affect the world's state?
- **Initial State:** How does the world look like initially?
- **Goal State:** What is the desired state of the world?
- **Planning Heuristics (User's Advices):** Advices from the user on how the plan can be computed.
- **Hierarchical Structure:** Information on how the problem can be broken down into smaller (and easier) problems.

In Section 3 below, we describe how these types of planning knowledge can be easily described using a set of pre-defined ontological constructs provided by our framework.

2.2 Ontological vs Goal-based Reasoning

From a logical perspective, both planning and ontological reasoning boil down to, and can be accomplished by, the task of proving an entailment. In the case of planning, the solution to the planning problem (i.e., a valid plan) can be considered to be an existential proof for the following entailment:

$$KB \models \exists p\, Valid(p) \wedge AchieveGoal(p)$$

where KB is the knowledge base representing the planning problem and $p = [a_1, a_2, \ldots, a_N]$ is the plan the user is looking for. In the case of ontological reasoning, subsumption and instance checking boil down to checking:

$$KB \models C \sqsubseteq D \quad or \quad KB \models a : C$$

respectively.

[1] By simple we mean planning problems that are deterministic (i.e., all actions have deterministic outcomes), instaneous (i.e., actions are assumed to have no duration), linear (i.e., no concurrency), and static (i.e., the environment's dynamics remain static throughout the planning cycle).

From a more technical point of view, however, these two types of reasoning are very different in nature and, as a result, most knowledge representation and reasoning formalisms typically support just one type or the other. Traditionally, planning is supported by rule-based formalisms that are based on Logic Programming (e.g., Prolog) while ontological reasoning are supported by object-oriented formalisms that are based on Description Logics (e.g., Ontologies).

Many practical ontology-driven applications, however, often require both types of reasoning and, as a consequence, there has been a great deal of interest in combining planning and ontological reasoning into a single formalism. Unfortunately, as explained in (Hitzler and Parsia, 2009), this is an inherently non-trivial task. First, combining the two formalisms leads to semantic-related issues because rule-based formalisms typically assume a closed-world model while description logics based formalisms assume an open world model. Second, adding language features from one formalism to the other often result in an undecidable language. In the next section, we briefly describes how existing approaches cope with these challenges, and discuss the pros and cons of each approach.

2.3 Existing Works on Integrating Planning into Ontology-driven Applications

Generally speaking, existing works on integrating planning into ontology-driven applications can be divided into three main approaches: Language Modification, Parallel Modelling, and Translation. In the first approach, the underlying language (i.e., Description Logics) is modified or extended to support rule-based reasoning. In the second approach, application knowledge are described in ontologies, while planning-related information are described separately in a rule-based language. In the third approach, planning-related knowledge are described using an ontology, alongside with other application knowledge, and translated into an executable rule-based planning program. We describe these approaches in more details below.

2.3.1 Language Modification Approaches

Because ontological reasoning is a feature of ontologies, and planning is a feature of rule-based formalisms, it is a fair question to ask if rules and ontologies can be reasonably combined to produce a more or less unified language in which both rule-based and ontological reasonings are dually supported in a seamless way. A lot of work in this direction have been reported, and readers who are interested in this topic are refered to (Hitzler and Parsia, 2009) for an overview, and (Horrocks et al., 2004), (Grosof et al., 2003), and (Motik and Rosati, 2008) for some better known example approaches. Here, we will focus our discussion instead on the pros and cons of such an approach.

Briefly speaking, the main advantage of a language modification approach is that of theoretical elegance. If successful, such a framework can serve as a unifying formalism that combines features from both rules and ontologies, two well-established knowledge representation and reasoning formalisms. The main disadvantage of this approach, however, is that it is inherently difficult, and success has been very limited so far (Hitzler and Parsia, 2009). In addition to the semantic (i.e., open world vs closed world) and complexity (i.e., decidable reasoning algorithms) issues mentioned in the previous section, modifying or extending a language often entails several other important tasks. First, adequate tooling support will need to be provided for the new language. This includes efficient reasoner implementations (assuming the new language is decidable) and effective editors for authoring models in the new language. Second, adequate experience reports will also have to be provided. This includes, among other things, case studies showing how such a formalism can be applied to solve practical real-world problems.

Given the difficult theoretical challenges above, and given the fact that most works in this direction are still in their early stages, it can be seen that language-based approaches, while theoretically rewarding and important, also have some disadvantages when considered as a mean for bringing planning to ontology-driven applications.

2.3.2 Parallel Modelling Approaches

Another popular approach to bringing planning to ontology-driven applications is the "parallel" approach in which planning and application knowledge are kept separated in two parallel worlds: planning-related knowledge are described in a (rule-based) planning language, while other application knowledge are described in ontologies. Integration is done by querying the ontologies for the list of available planning actions, and perhaps their pre-conditions, and executing the planning program using these actions.

Several works from the ontology-driven workflow composition community have been reported to follow this approach. (Bernstein et al., 2005), (Žáková et al., 2008) and (Diamantini et al., 2009), for example, de-

scribe three ontology-driven frameworks that employ a planner to compose data mining (DM) workflows (i.e., applications) from individual DM algorithms. In these frameworks, DM algorithms – each of which is considered a planning action – are ontologically described using an ontology. A planning program would then query this ontology to extract the list of available algorithms, together with other relevant information such as their pre-conditions, etc., and then compose the workflows by putting these algorithms together.

While these frameworks have successfully demonstrated the practical feasibility of employing a planner to solve ontology-driven planning problems, there is one important disadvantage in their approach. Because planning-related knowledge was encoded in a planning language instead of ontology, reusability and interoperability of this knowledge is therefore reduced[2]. To some degrees, this weakness defeats the purpose and benefits promised by an ontology-driven approach.

2.3.3 Translation Approaches

Another promissing approach to bridging planning and ontological modelling is to 1) use a dedicated ontology to describe all planning-related information, and then 2) use an automated translator to translate these information into an executable planning program that can be executed in a rule-based framework.

The first part of this approach – describing the planning problem using a dedicated planning ontology, is represented by the work reported in (Rajpathak and Motta, 2004). In this paper, the authors presented a generic planning ontology that can be use to describe various types of planning problems. This ontology contains all the basic planning concepts such as Goals, Actions, Agents, Planning Constraints, Pre/Post Condition, Reward/Cost, Optimization Criteria and Preferences, Temporal ordering of Actions, etc.[3]

The second part of this approach – translating the ontological description of the planning problem into an executable planning program and executing it in a planning framework, has no existing representation in the literature however. Despite its apparent ability to *describe* advanced planning problems, the work reported in (Rajpathak and Motta, 2004) does not discuss how the planning problems, once described, can be translated and executed in a planning engine to produce the desired results. In doing so, a few important practicality issues are to be expected. The first issue is that of translatability: How do one ensure that all planning problem descriptions are translatable in to equivalent rule-based programs? Because ontology (i.e., Description Logics) and rules are two different languages, there is the possibility that an ontological description can not be translated into an equivalent rule-based version. The second isssue is that of effectiveness: Assuming that the planning problem is translatable into a rule-based program, how does one ensure that it can be executed in an effective manner. As already discussed in Section 2.1, real-world problems often result into planning problem that are beyond even the most capable planners, and an effective planning framework needs to provide support for coping with this complexity.

In the remaining parts of this paper, we address these practicality issues and present a complete translation-based, ontology-driven planning framework for ontology-driven applications.

3 A PRACTICAL ONTOLOGY-DRIVEN PLANNING FRAMEWORK

In this section, we propose an ontology-driven planning framework that completes and realizes the translation approach discussed in Section 2.3.3 above. Figure 1 illustrates the basic ideas of this framework.

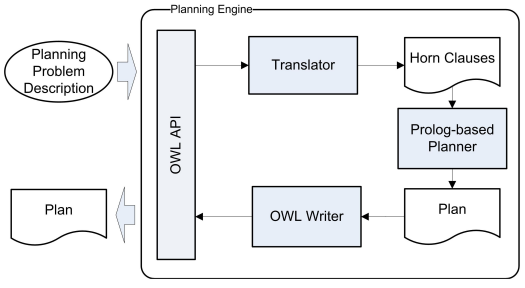

Figure 1: An ontology-driven planning engine. This engine's design illustrates the essence of our translation approach to integrating planning into ontology-driven applications. Detailed descriptions of the components shown are found in Section 5 below.

[2](Diamantini et al., 2009) encoded their planning program in a generic and framework-independent planning language called PDDL. This language, one could argue, could be considered a form of ontology in itself, but for this discussion, we will restrict out interpretation of ontology to just the Description Logics-based ontology language defined by the Semantics Web community.

[3](Gil et al., 2000) also describes a dedicated planning ontology called PLANET. This ontology, however, was intended to be a language for describing "plans" as opposed to "planning problems" (the first is a solution of the second), and therefore is not directly related to our discussion.

To use our framework, ontology-driven application designers would use the ontological constructs that are defined by our Planning Ontology to describe their planning problems into a knowledge base. The framework will translate this problem description into an equivalent executable program in Horn Logic, execute it in a Prolog-based planning engine, and return a valid plan back to the user.

In Sections 4 and 5 below, we will go through in-depth examples to illustrate how planning problems can be described using the ontological constructs provided by the Planning Ontology, how these descriptions are translated into executable programs and executed, as well as how the practicality issues such as translatability, effectiveness and semantical conflicts are addressed. For now, we would like to explain how our framework would typically fit into the overall design of an ontology-driven application, and the practical benefits that it offers over existing approaches.

3.1 Overall Application Architecture

Figure 2 illustrates how our proposed framework could be used to build ontology-driven planning applications in a reusable and practical way.

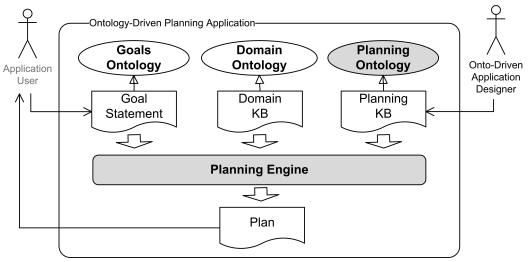

Figure 2: A generic architecture for building reusable ontology-driven planning applications. The Planning Ontology is described in details in Section 4. The Planning Engine's design is shown in Figure 1.

Using this architecture, an ontology-driven planning application would organize its application knowledge into three separate parts.

First, a "Goal Ontology" is used to describe the different goal statements that the user would provide as inputs to the application. For the intelligent trip planner example, these goal statements could look something like "Find the fastest way to get from University of Guelph to the KEOD2011 conference" or "Find a way to get from Paris to Nantes without flying", etc.

Second, a "Domain Ontology" is used to describe all the relevant concepts of the domain in which the application operates. In the trip planner application, for example, the Domain KB will contain the descriptions for all the flights, bus and train routes, together with supporting concept such as airports, train stations, bus stops, fares, cities, hotels, etc.

Finally, all planning-related knowledge needed to drive the planner and build the plans is captured in the Planning KB. This KB uses the ontological constructs provided by the Planning Ontology to specify, among other things, which domain concepts can be considered a planning action, what their pre-conditions are, how the world will changes in response to each action being performed, how the problem can be best solved, etc.

3.1.1 Practical Advantages

Our proposed framework and its architecture offer four main advantages over existing approaches.

First, because planning-related knowledge are kept separated from the domain ontology and KB, this ontology can be developed and maintained independently from the planning application itself. This indepedence not only makes the domain ontology simpler to develop and maintain, but also makes it application and purpose-independence, and therefore much more resuable for future applications.

Second, because planning-related knowledge is described in an ontology instead of a framework-specific planning language, the resulting description of the planning problem is completely independent from the underlying planning framework, and therefore can be processed, translated, and executed by any planning framework that are capable of processing ontology-backed knowledge bases.

Third, by taking a translational approach, as opposed to a language modification or extension approach, our framework is able to make use of existing and mature theoretical frameworks (Horn Logic) and technologies (Prolog programming language) to provide seamless goal-based reasoning capability in ontology-driven applications.

Lastly, using our framework, the ontology-driven application designer can continue to think and work in the ontological modelling environment that he or she is already comfortable with. Instead of having to learn either a rule-based planning language or a new extension to the ontological modeling language, the application designer can simply decribe his planning problem in his familiar ontology editing enviroment, and have the framework handles all the mappings for him.

4 DESCRIBING PLANNING PROBLEMS

In this section, we dive into the various ontological constructs that are offered by the Planning Ontology, and show how planning problems can be effectively described in a fashion that is completely independent from the underlying planning framework.

4.1 Basic Planning Constructs

The first group of language constructs in the Planning Ontology are those that are needed to describe the basic ingredients of the planning problem. Figure 3 will be used as illustrate examples for these contructs.

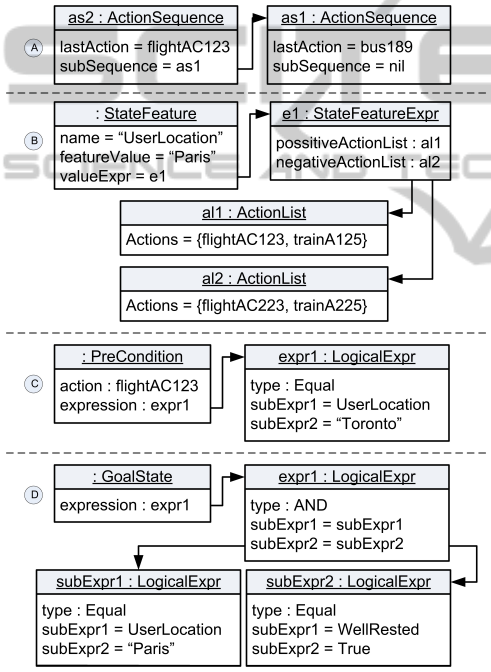

Figure 3: Examples showing how the basic ingredients of the planning problem can be described using the Planning Ontology.

4.1.1 Actions

Planning actions are described using the *Action* concept in the Planning Ontology. In the trip planner application, for example, all flights, bus routes, train routes, as well as *walk* and *getCoffee* can all be marked as actions. In particular, we assume, for the purpose of our discussion below, that flight AC123 is Paris-bound flight, train A125 is a Paris-bound train, while flight AC223 and train A225 are flight and train ride in the opposite directions.

4.1.2 Environment's Dynamics

We follow the knowledge specification approach described in (Reiter, 2001) and describe the world's state using a pair of concepts: The ActionSequence concept and the StateFeature concept[4].

The ActionSequence concept is used to represent a sequence of planning actions. "Take the 189 bus to the airport and then board the AC123 flight to Paris", for example, is a sequence of 2 actions, and can be represented as shown in Figure 3.A.

The StateFeature concept is used to represent a particular aspect of the world. The traveler's location, for example, is one such feature. As the traveler moves around in her trip, her location changes, and the StateFeature concept can be used to describe how the traveler's location change its value in response to the various actions the traveler can take. As illustrated in Figure 3.B, the traveler's location will take on the value of the flight's destination (e.g., "Paris") if her last action is either flight AC123 or train A125 (i.e., two Paris-bound actions). These two actions are parts of the PositiveActionList because, if taken, they cause the StateFeature to take the specified value. Flight AC223 and train A225, on the other hand, are on the NegativeActionList because, if taken, they cause the StateFeature to *not* take the specified value. In Section 5 below, we will explain how this form of description is translated into a formal logical statement[5]. For now, it suffices to note that the PossitiveActionsList property needs to contain all the actions that would cause the state feature to take the value specified, and the NegativeActionList property need to list all the actions that would cause the state feature to change its value away from the specified value. All other actions that have no effects on the feature value, such as getCoffee or takeABreak, do not need to be included in the description of state feature.

4.1.3 Actions' Pre-conditions

Actions' pre-conditions are described using the PreCondition concept from the Planning Ontology. As illustrated in Figure 3.C, the precondition for taking a flight is that the traveller is located in the same city as the flight's departure airport. More complex preconditions expression involving multiple state features can be built up using the LogicalExpression concept and its sub-concepts.

[4]These are called *Situation* and *Fluent* in Reiter's. We use a different terminology to make it more relatable for application designers who might not be familiar with Situation Calculus.

[5]This is called a Successor State Axiom in Reiter's.

4.1.4 Intial State and Goal State

The initial and goal states are also specified using the LogicalExpression concept similar to the way precondition expressions are specified. Figure 3.D show an example goal state specification that requires the user to be both in Paris and well-rested.

4.2 Model Closure Constructs

One of the standard cautions one has to take when integrating goal-based reasoning with ontological modelling is the open-world vs closed-world assumptions conflict. Goal-based program's KBs are closed-world models (a fact can be assumed to be false if it has not been stated otherwise), while ontological models are open-world models (a fact cannot be assumed false unless it was explicitly asserted). For the purpose of describing planning problems, however, this is not a practical issue – the appplication designer just have to picture a closed world in his mind when describing his planning problem. To help make it mentally explicit for the designer however, the Planning Ontology provides ontological contructs that he can optionally insert into the planning KB to logically (or mentally) "close" the model down. The statement "No other planning actions are available", for example, when inserted into the KB, has the effect of finalizing the list of already asserted planning actions, and hence provides a mental closure to the model.

4.3 Planning Effectiveness Constructs

4.3.1 Planning Heuristics

As we discussed earlier, real-world planning problems are often too complex for even the most advanced planners to solve exhaustively and an effective and practical planning framework must provide convenient facilities and mechanisms for the application designer to provide heuristic insights to the planner. In our framework, this can be done via a mechanism called *partial programming*, also described in (Reiter, 2001). The idea is that, instead of computing plans from scratch, the planner would start from a partial template that the system designer has provided. Because this template contains all the heuristic advices from the designer, computing plans from this template will be much faster and efficient than computing one from scratch.

In the trip planning problem, for instance, one possible heuristic advice the system designer might want to provide the planner is "If the traveller is at a hub airport (where the number of connecting flights is larger than other smaller airports), then she should either try to find a direct flight from that airport to the destination, or find a flight to go to another hub airport". This kind of advice helps the planner to avoid sending the traveller from a hub airport to a local airport where she would get stuck and have to spend extra time to get back out.

We will describe how this heuristic advice can be easily provided to the planner after explaining the list of partial programming constructs that are provided by the Planning Ontology.

- **ActionTemplate:** From a conceptual perspective, this concept is used to represent a partial template from which a full plan can be computed. From a procedural perspective, this concept serves as the equivalence of a procedure from a programming language, except that this procedure might only be partially specified. Each template has a body which is a TemplateExpression (described below).
- **TemplateExpression:** This concept is use to describe the procedural details of an ActionTemplate. There are 4 main kinds of TemplateExpression:
  - **SequentialCompostion:** This concept is used to express the fact that a procedure is consist of two sub procedures, one is to be executed before the other.
  - **IfThenElse:** This concept is used to express the fact that a procedure is consist of two exclusive branches. If a certain condition hold, the first branch is executed. Otherwise, the other branch is.
  - **WhileLoop:** This concept is used to express the fact that a procedure is consist of a sub procedure that need to be repeatedly executed as long as a certain condition holds.
  - **ChoicePoint:** This concept is used to represent the fact that a procedure is consist of two exclusive sub procedures, and the planning engine has the flexibility of selecting either one. Using this construct, the application designer can convey to the planning engine that, instead of having to consider all possible actions, it can narrow its choices to just the subset of sub-plans specified in the choice point. The more insight the designer has about the workflow composition problem, the more choice points he will put in the template, and the less work the planner has to do.
  - **Action:** This is the simplest template expression possible. It represents the fact that fact a procedure is consists of a single planning action.

Using these partial programming constructs, the example heuristics advice above can be provided to the planner as shown in Figure 4.

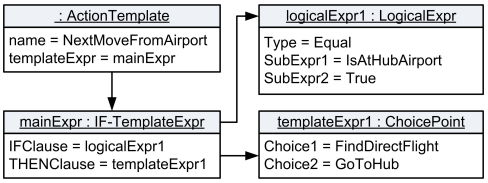

Figure 4: An example showing how planning heuristics can be supplied to the planner using partial programming constructs.

4.3.2 Hierarchical Composition

Reaching the goal using only primitive actions often requires very long plans (plans with very long sequences of actions) to be computed, and is often infeasible for practical real-world problems. One way to get around this problem is to provide the planner with some additional information about how the problem can be broken down into smaller problems so that it can be solved more effectively. The main idea is to quickly reach the goal state with a few large steps first, and then gradually flesh these large steps out into a more fine-grained plan.

Our Planning Ontology provides support for hierarchical planning through a language construct called MacroAction. Using this construct, the system designer can mark an arbitrary ActionTemplate (i.e., partial procedure) as a planning action, complete with its own preconditions and effects. The planner can then use these macro actions, alongside with other regular actions in a seamless manner, to quickly compute high-level plan satisfying the objective. Once such plan has been successfully found, the macro action can be iteratively fleshed out into concrete subplans.

5 TRANSLATING AND EXECUTING PLANNING PROBLEM DESCRIPTIONS

In this section, we describe how the each of the language constructs discussed in Section 4 above are mapped into an Horn clauses to make an executable rule-based planning program, and provide an intuition on why this mapping is alway possible. We will also briefly describe how the translated program can be executed using a custom Prolog-based planner to produce valid plans.

5.1 Translation

As described in Figure 1, our framework queries the ontolgy knowledge bases and generate Horn clauses from the asserted planning knowledge. In this section, we will use Prolog's syntax and notation to describe these Horn clauses.

5.1.1 Translating Basic Planning Constructs

First, for each domain entity that are marked as a planning action, we generate a clause of the following form:

```
action(actionConcept).
```

From our trip planner example, we have:

```
action(flight(FlightNo, Src, Dst)).
action(train(TrainNo, Src, Dst)).
action(getCoffee).
action(takeABreak).
```

Second, for each StateFeature assertion in the model, we generate a rule of the form:

```
featureName(X, [A | ActionSequence]) :-
  A = posAction1; ...; A = posActionM;
  featureName(X, ActionSequence),
  not A=negAction1, ..., not A=negActionN.
```

where semicolons mean OR in Prolog, and commas mean AND. Also, M is the number of actions in the PositiveActionList, and N is the number of actions in the NegativeActionList. The UserLocation state feature assertion, for example, is translated as:

```
userLocation(paris, [A | ActionSequence]) :-
  A = flightAC123; A = trainA125;
  userLocation(paris, ActionSequence),
  not A=flightAC223, not A=trainA225.
```

In English, this rule says that the UserLocation feature will take on the value "Paris" if either of the following is true: a) The last action in the action sequence is either flight AC123 or train A125 (both of which take the traveler to Paris), or b) The traveler was in Paris before the last action was taken, and that action is not flightAC223 and trainA225 (both of which would take the traveler out of Paris). All other actions, such as getCoffee, or takeABreak, have no effect on the value of this feature.

Third, for each precondition assertion, we would generate a clause of the form:

```
poss(action, ActionSequence):-
logicalExpression.
```

e.g.:

```
poss(flight(FlightNo,Src,Dst),ActionSequence)
:-userLocation(X,ActionSequence),X = Src.
```

which says that it is possible to select a flight as the next action if the user is located in the departure city of the flight.

Fourth, the intial and goal state description are translated in a straight forward manner. For the trip planning application, we have:

```
userLocation(guelph, []).

goalState(AS):-
  userLocation(paris, AS),
  wellRested(AS).
```

Finally, all model closure assertions are simply ignored by the translator.

5.1.2 Translating Heuristic Advices

While descriptions of the basic planning ingredients are translated directly into Prolog statements, user heuristic advices (i.e., instances of the ActionTemplate concepts) are translated into an intermediate language. We explain this translation below.

For each instance of the the ActionTemplate in the ontology, we generated a text string of the following form

```
PROCEDURE Name { Body }
```

where Name is the name of the ActionTemplate, and Body is a text string representing the TemplateExpression of the ActionTemplate. Recall from Section 4.3.1 that TemplateExpr can take several forms. We shows the corresponding body texts that get generated for each form of TemplateExpr below.

Table 1: Translating Template Expressions. Uppercase letters are keywords, while a, b and c are the subexpressions of the TemplateExpressions.

TemplateExpr	Generated Text
Seq. Comp.	a ; b
IfThenElse	IF c THEN a ELSE b ENDIF
WhileLoop	WHILE c THEN a ENDWHILE
ChoicePoint	a OR b
Simple Action	a

The ActionTemplates shown in Figure 4, for example, are translated into the following text:

```
PROCEDURE FindDirectFlight {
   // Some code goes here
}

PROCEDURE GoToHub {
   // Some code goes here
}

PROCEDURE NextMoveFromAirport {
   IF (IsAtHubAirport)
   THEN
       Anonymous1
   ENDIF
}

PROCEDURE Anonymous1 {
   FindDirectFlight
   OR
   GoToHub
}
```

Once all the heuristics advices (i.e., ActionTemplates) have been translated into intermediate code, the translation engine then translate this code into a Prolog program segment[6]. This translation is done recursively using an expansion algorithm, whose psuedo code is shown below:

```
Expand(expr)
{
  structure = expr.GetTopLevelStructure()
  swictch(structure){
  case (a ; b):
    return GenPrologStr(Expand(a),
                        ",", Expand(b))
  case (IF c THEN a ELSE b ENDIF):
    return GenPrologStr(eval(c), ",",
                        Expand(a),
                        ";",  Expand(b))
  case (a OR b)
    return GenPrologStr(Expand(a),
                        ";", Expand(b))
  case (a)
    return GenPrologStr(Expand(a))
  }
}
```

Essentially, this algorithm recursively expands the TemplateExpression into a (long) Prolog rule representing the a structure of a plan. This structure contains all the actions that needed to be done, with some "OR" operators inserted in between to allow the planner to make its own choice.

The intermediate code discussed earlier, for example, is recursively expand into Prolog template statement as follows:

```
Step1:
(isAtHubAirport(AS), anonymous1 ) ; true.

Step2:
(isAtHubAirport(AS),
    (findDirectFlight ; goToHub)) ; true.

Step3:
...
```

[6]This translation engine design borrows the macro expansion technique used by the GOLOG interpreter described in (Reiter, 2001).

5.2 Execution

Once the basic ingredients of the planning problem description has been translated into a Prolog KB as described in Section 5.1.1 and the heuristic advices have been translated into an intermediate text file and then into a Prolog template statement as described in Section 5.1.2, the translated codes are merged together into a single Prolog program, and can be loaded for execution in Prolog.

The execution is handled by the predicate:

```
?- findPlan(Plan, templateStatement).
```

where Plan is an "output variable" and templateStatement is the Prolog template statement discussed above. If findPlan succeeds, Plan is initialized (unified) with a sequence of actions conforming to the templateStatement, except that all the branching represented by the ";" operator in templateStatement are replaced with a single straight path.

5.3 Ensuring Translatability

We conclude this section with a discussion regarding the translatatability of planning problem descriptions.

As with any translation approaches, one of the main theoretical questions that is of importance to our proposed approach is that of translatability – How do one ensures that the planning problem descriptions created by the ODWC system designer are always translatable into an executable planning program in Horn Logic? To answer to this question, a few observations are in order.

First, a well-defined ontology can be thought of as a form of language – The list concepts its provides constitute the vocabulary of the language, while the roles it defines dictates the ways in which the vocabulary can be combined together to form statements. Second, by carefully controlling the list of the concepts and roles in the ontology, we can restrict or control the types of statements one can express using the ontology.

With these observations in mind, one could see that by being very selective and careful with the languague constructs in the Planning Ontology, we can ensure that all possible workflow compostion problem descriptions are translatable to executable planning programs in Horn Logic. This, in fact, is the main intuition behind our approach. Figure 5 provide a visual illustration for this intuition.

In a future article, we will report whether or not a formal proof of this intuition is possible. This is still a work-in-progress, but our impression is that by carefully analyzing the structures of these language constructs, one would be able to prove, via structural induction, that all planning problem expressed via our Planning Ontology are translatable into equivalent executatable Horn programs.

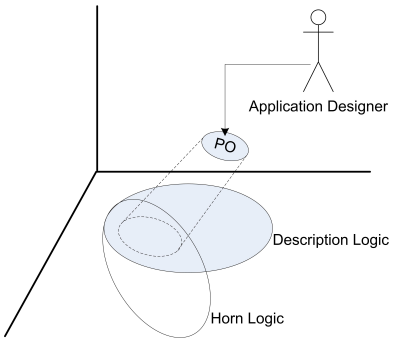

Figure 5: The primary intuition behind our proposed approach: the Planning Ontology acts as a restrainer that helps ensure the description of the planning problem always falls into a sub area of Description Logics that is translatable to an executable program in Horn Logic

6 SUMMARY

While Ontology and Description Logics do not provide built-in support for the type of rule-based reasoning needed to do logic-based planning, they are fully capable of *describing* planning problems. We believe that harnessing this descriptive power to build an ontology-driven planning framework offers several practical advantages over existing language extension or adhoc integration approaches. First, it allows planning to be integrated into ontological modelling in a seamless and transparent fashion: Ontology-driven application designers do not need to pickup a new language or formalism in order to bring planning capabilities to their applications. All they need to do is to simply describe the planning problem using their familiar ontology modelling tools, and the framework takes care of the rest. Also, because planning knowledge are described in an ontology instead of a framework-specific language, they are not only much more reusable, but also can be processed and executed by more planning frameworks.

In this paper, we provided a survey of existing approaches to integrating goal-based reasoning and ontological modeling, and provided some arguments for an ontology-driven planning framework. We also provided a detailed discussion of our proposal for such a framework and described how it can be used by ontology-driven application designers to build intelligent knowledge-based systems in a transparent and reusable way.

The next objective we have for the framework is a formal proof on the translatability of planning problem descriptions. As discussed earlier in the paper, we believe that this is possible with a careful analysis of the structure of the language constructs provided by the Planning Ontology.

For a further information on our framework, the reader is invited to visit our website at http://ontology.socs.uoguelph.ca.

REFERENCES

Bernstein, A., Provost, F., and Hill, S. (2005). Toward intelligent assistance for a data mining process: An ontology-based approach for cost-sensitive classification. *IEEE Transactions on Knowledge and Data Engineering*, 17:503–518.

Diamantini, C., Potena, D., and Storti, E. (2009). Ontology-driven kdd process composition. In *Proceedings of the 8th International Symposium on Intelligent Data Analysis: Advances in Intelligent Data Analysis VIII*, IDA '09, pages 285–296, Berlin, Heidelberg. Springer-Verlag.

Gil, Y., Gil, A., and Blythe, J. (2000). PLANET: A Shareable and Reusable Ontology for Representing Plans. In *Proceedings of the AAAI Workshop on Representational Issues*.

Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. (2003). Description logic programs: combining logic programs with description logic. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 48–57, New York, NY, USA. ACM.

Hitzler, P. and Parsia, B. (2009). Ontologies and rules. In Bernus, P., Blazewics, J., Schmidt, G., Shaw, M., Staab, S., and Studer, R., editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 111–132. Springer Berlin Heidelberg.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., and Dean, M. (2004). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, World Wide Web Consortium.

Motik, B. and Rosati, R. (2008). Reconciling description logics and rules. *J. ACM*, 57:30:1–30:62.

Rajpathak, D. and Motta, E. (2004). An Ontological Formalization of the Planning Task. In *Proceedings International Conference on Formal Ontologies in Information Systems (FOIS'04)*, Torino, Italy.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, Massachusetts, MA, illustrated edition edition.

Russell, S. J. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall.

Žáková, M., Křemen, P., Železný, F., and Lavrač, N. (2008). Using ontological reasoning and planning for data mining workflow composition. In *SoKD: ECML/PKDD 2008 workshop on Third Generation Data Mining: Towards Service-oriented Knowledge Discovery*.