A PRACTICAL PLANNING INTEGRATION FRAMEWORK
FOR ONTOLOGY-DRIVEN APPLICATIONS

by

Huy Pham

A Thesis
presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of
Doctor of Philosophy
in
Computer Science

Guelph, Ontario, Canada
© Huy Pham, August, 2013
ABSTRACT

A PRACTICAL PLANNING INTEGRATION FRAMEWORK
FOR ONTOLOGY-DRIVEN APPLICATIONS

Huy Pham
University of Guelph, 2013

Advisors:
D. A. Stacey & A. Ferworn

Despite the many clear advantages that ontology has to offer as a standard-
ized knowledge representation language, many intelligent system developers
still find it difficult to jump on the band wagon and represent all their applica-
tion knowledge using ontology. This difficulty and hesitation stems primarily
from the fact that, while most ontology languages provide native support for
reasoning about the domain’s structures, they do not provide adequate support
for computational planning – the kind of reasoning used by many intelligent
systems to derive their purposeful behaviors.

To overcome this challenge, a lot of work has been done to discover a prac-
tical way to seamlessly incorporate planning into ontology languages. As it has
been well-established in the literature however, this is a very challenging task
from both a theoretical and practical standpoint, and many of the reported
works in this direction either have had very limited success, or have been done
in ad hoc and less reusable manners.

In this thesis, we report our pursuit of a new approach to integrating
planning into ontology-driven applications. This approach promises to overcome the difficulties faced by many of the existing approaches. In addition to producing a reusable and extensible framework for doing computational planning in ontology-driven applications, our pursuit also raises and answers some interesting ontology research questions that could have potential impacts on several application domains beyond the integration of planning and ontological modeling.
Acknowledgements

As I reflect back on this long and eventful journey, my mind is filled with an overwhelming sense of gratitude. Over the seemingly countless number of years that it took me to come to this point, I have become highly indebted to so many people around me.

Dr. Deborah Stacey, my thesis advisor. Unquestionably, this whole journey would have been a mission impossible without Deb’s guidance, support, and encouragement over the years. I would like to thank her for the countless number of memorable whiteboard discussions, for the many free trips to the coffee shop, and for the many beautiful moments we shared. Most importantly, I would like to thank her for the motherly care and teaching that have firmly shaped who I am both as an academic and as a person in general. I am forever indebted to her.

Dr. Qusay Mahmoud, my initial advisor. It was Qusay who took me into the program and set me off on this memorable journey. The time I spent under his wing was filled with beautiful moments. I would like to thank him for his guidance, support, and for the many things that he has done for me that I can never forget.

Dr. Alexander Ferworn, my thesis co-advisor. Alex has always been there for me through the ups and downs of the entire journey. His help, support, and earthly advice helped put important things into the right perspectives. I would like to thank him for all his help and support, and for the invaluable mentorship he has given me over the years.
Dr. David Calvert, my advisory committee member. Above and beyond providing me with important insights and clarifications that helped solidify and crystalize many important ideas in this thesis, Dave have always been there for me through the entire process. I would like to thank him for his help, for being such a kind person, and for the many entertaining hours we shared at the lunch table with others.

Dr. Judi McCuaig, my thesis examiner. Judi’s highly constructive questions and suggestions have brought clarification to many ideas in this thesis and made it a lot stronger. I would like to thank her for all her kind feedbacks and suggestions.

Dr. Stefan Kremer, my examining committee chair. Stefan’s calm and friendly present in the room during my thesis defense was a big help that I truly appreciated. I would also like to thank him for his very detailed notes of the defense, which helped me a lot during the final revisions after the defense.

Dr. Kamran Sedig, my external examiner. I would like to thank Dr. Sedig for serving as my external examiner, and for his many thoughtful questions which helped bring clarifications to many of the arguments in my thesis.
Contents

1 Introduction 1

2 Background 9
   2.1 Ontology .............................................. 9
   2.2 Computational Planning ................................ 12
   2.3 Situation Calculus as a logical planning formalism .... 14
   2.4 Planning in Ontology-Driven Applications ............ 20
      2.4.1 Planning vs Ontological Reasoning .............. 22
      2.4.2 Existing works on integrating planning into ontology-
            driven applications .......................... 22
   2.5 Web Service Composition .............................. 28

3 A Practical Planning Integration Framework for Ontology-
   Driven Applications 35
   3.1 Rationales for a Translation Approach ............... 36
   3.2 Addressing the Primary Challenges .................. 39
      3.2.1 Representability ................................ 39
      3.2.2 Translatability and Constrainability .......... 43
Chapter 1

Introduction

Many of today’s intelligent systems are knowledge-intensive in nature. This intensity, which can manifest itself both in terms of volume and in terms of complexity, has placed some extra requirements on the system developers [39]. In addition to the usual intelligence-related issues, today’s system developers also have to pay close attention to the effective representation of their application knowledge. Because larger and richer knowledge bases are much harder and more expensive to construct and maintain, the reusability, richness and effectiveness of their representations play a critical role in the overall success of the system.

Driven by these new requirements, intelligent system developers have been looking into representing their application knowledge in ontology languages such as the Web Ontology Language (OWL) [2, 42, 76], an expressive knowledge representation (KR) formalism based on Description Logics [3]. As compared to other KR formalisms and languages, OWL offers several important advantages, among which are expressiveness, modularity, reusability, and au-
tomated reasoning capabilities [39].

All these important advantages would have made OWL an ideal KR language for knowledge-based and knowledge-intensive intelligent systems. As it has been reported in the literature, however, this has not always been the case. While OWL provides efficient support for the types of ontological reasoning described above, it does not provide native support for planning, a reasoning mechanism that is needed by many intelligent systems. As opposed to ontological reasoning, which uses the logical definitions of the concepts and objects in the model to make inferences about their taxonomic (i.e. “class-subclass” and “instance-of”) relationships, planning uses the logical descriptions of actions to search for a valid plan that could accomplish a given set of goals or objectives. Also, in ontological reasoning, the environment is considered to remain static for the entire duration of the reasoning process (that is, no state change occurs during the reasoning process), whereas in planning, the environment changes its state with every action considered.

Because many intelligent systems derive their intelligent and purposeful decision-making capability via a goal-based mechanism, this lack of native support for planning in ontology languages presents a serious challenge for intelligent system developers looking to take advantage of an ontology-driven approach. As a result of this, there has been a lot of interest in finding a practical way to incorporate planning capability into ontology, and a lot of work has been reported in this area.

Generally speaking, existing works on integrating planning into ontology-driven applications can be divided into two main approaches: Language modification and parallel modeling. In the language modification approach, the
underlying language (i.e. Description Logics) is modified, extended, or restricted to support planning. As will be described in more detail in Chapter 2 below, the primary rationale behind this approach is that since planning has been well supported by rule-based formalisms such as Logic Programming [63], one could potentially integrate planning capability into DLs by incorporating language features from these formalisms. This approach, if successful, would be theoretically elegant and valuable, as it would have unified, at least to some degrees, two of the most important and well-established KR formalisms. As it has been well-established in the literature, however, reconciling these two formalisms is a highly non-trivial task that can easily lead to several difficulties, both theoretically (e.g. decidability, semantics mismatch, etc.) and practically (e.g. expressiveness, tooling and support ecosystems, user acceptance, etc.). Consequently, many of the existing works of this nature have had very limited success to date [53].

In the parallel modeling approach, application knowledge is described in ontologies, while planning-related knowledge is encoded separately as planning programs in a rule-based language. At execution time, the ontologies are queried for relevant information, and the planning program is executed with this information in a rule-based framework. As compared to the first approach, this approach has the advantage of practicality — existing tools and technologies can be employed to solve ontology-driven planning problems without having to wait for new languages or frameworks. The primary disadvantage of this approach, as will be explained in more details in Chapter 2 below, is that of reusability. Because planning-related knowledge is encoded in a (usually application-specific) planning language instead of ontology, the reusability and
interoperability of this knowledge is therefore reduced. To some degree, this weakness undermines or even defeats the purpose and benefits promised by an ontology-driven approach.

In this thesis, we propose a new approach for integrating computational planning into ontology-driven applications (i.e. knowledge-based applications in which the primary knowledge representation language used is an ontology language such as the Web Ontology Language (OWL) [2, 42, 76]). Using our proposed approach, planning-related knowledge are represented directly in ontologies, alongside with other application knowledge, and then translated into executable planning programs at run time. This translation approach, illustrated in Figure 3.1, is based on the basic observation that, while ontology does not provide adequate support for planning, it may be fully capable of describing planning problems. As such, if we can harness this existing descriptive power and model all application knowledge in ontologies, we could perhaps avoid the reusability issue faced by the parallel modeling approaches above, and provide a more practical alternative to the language modification approaches.

During our pursuit, several important questions needed answering. In particular:

1. **Representability – Is it possible to describe or model planning problems in ontology?** Traditionally, planning problems are modeled in rule-based languages such as Horn Logic (HL) [70]. As shown in Figure 1.1 below, this language is not a proper subset of Description Logic, and so a probability exists for some planning knowledge that was expressible in HL become non-expressible in DL/ontology. Also, while HL assumes a
Figure 1.1: Description Logic (DL) and Horn Logic (HL) as two decidable subsets of First Order Logic (FOL). These two languages can be thought of as two different ways of attaining decidability via limiting the scope of the language. Due to the differences in the ways in which their scopes were limited, each of these two languages is well-suited for a different purpose – DL is well-suited for describing and reasoning about the structure of object-oriented worlds, while Horn Logic is well-suited for planning and hence problem solving.

closed-world model, DL assumes an open-world model. This semantic mismatch can also cause some expressibility issues.

2. Translatability and Constrainability – Assuming that a planning problem has been successfully described/modelled in an ontology, is it always possible to translate it into equivalent executable rule-based programs? DL is also a non-proper subset of HL, and so it is extremely important that we can find an effective way to ensure the users do not describe their reasoning problems in such a way that can not be translated into the language of HL.

3. Effectiveness and Reusability – If such a framework is feasible, will it be effective enough to handle non-trivial, real-world planning problems?
Also, can it be re-used across application domains? Finally, can it be extended to accommodate new requirements when they arrive? To be a practical alternative to existing approaches, such a framework would need to be effective enough to handle real-world planning problems. Also, to demonstrate a clear advantage over ad hoc approaches, this integration framework would need to be generic enough to be reusable across multiple application domains.

Based on the observations and intuitions that we will explain in details in Chapter 3, we are confident that a positive answer to each of the questions above is possible, and set out to prove the following statement:

**Thesis Statement** - *Planning in ontology-driven applications can be effectively supported via the use of a reusable and extensible translation integration framework.*

There are three major components to this claim – Feasibility, Effectiveness, and Reusability. As will be reported in detail in Chapters 3 and 4, we will demonstrate these characteristics by designing and constructing an actual integration framework, and then applying it to solve two different reasoning problems from two different domains. In particular, we demonstrate feasibility of the framework by constructing and applying it to build a simple trip planner, which helps its user get to a destination by picking the appropriate flights, bus and train routes. To demonstrate the framework’s effectiveness and reusability, we apply it to solve the problem of building an intelligent student advisor, which provides students with real-world course selection recommendations based on the student’s goal. Among these tasks, the feasibility
demonstration task is perhaps the most critical one, as it involves confirming the validity of two of our proposed techniques – The first one being a technique for representing HL statements in DL, and the second one being a technique for constraining the users to translatable DL descriptions in a transparent and non-intrusive way.

The primary contribution of our effort, therefore, is twofold. First, it provides an effective, reusable, and extensible integration framework that can be used by application developers to build ontology-driven intelligent applications in a more modular and reusable way. Second, it serves as a concrete illustrative example for the two techniques that we proposed for the feasibility task. As will be explained and elaborated more in Chapters 3 and 5, these observations and techniques could potentially be very useful for other areas of ontology research beyond the integration of planning and ontology modeling.

The contents and organization of this thesis is as follows. Chapter 2 discusses some background information on ontology and planning, and reviews existing approaches to supporting planning in ontology-driven applications. Chapter 3 discusses the rationales for a translation approach, the primary challenges that such an approach has to overcome, and the details of our proposal. Chapter 4 discusses our methodology, as well as two case studies in which we applied the proposed framework to build two different intelligent ontology-driven systems. Finally, Chapter 5 concludes the thesis and discusses the potential applications of the primary techniques that are discussed in this thesis.
(This page was intentionally left blank)
Chapter 2

Background

2.1 Ontology

The word Ontology comes from the Greek words *ontos*, which means “being”, and *logia*, which means “science” or “theory”, and often refers to a branch of Philosophy that studies, among other things, the different categories of things and existences, how these existences should be classified.

Within the field of Computer Science, ontology is a knowledge representation formalism that allows application and domain knowledge to be represented in a formal (i.e. machine understandable) and reusable way. Intuitively, an ontology is a collection of logical statements describing a certain world or domain [81][22]. These statements constitute what is often referred to as a formal conceptualization of the domain [46]. This conceptualization represents a shared understanding of the domain, and provides the vocabulary needed

\footnote{More precisely, an ontology is often defined as “a formal, explicit specification of a shared conceptualization”. [46]}
to describe or express different types of knowledge in that domain. An animal ontology, for example, is a machine-understandable model of the animal kingdom, and would contain the vocabulary needed to describe different facts about this kingdom. In particular, such ontology would contain formal and explicit definitions for all the important concepts that are found in the animal world, including:

- Various types of animals, such as Elephants, Tigers, Buffaloes, Carnivores, Herbivores, Omnivores, Mammals, Fish, etc. These are called the concepts of the domain.

- Various kinds of attribute that these animals can have, such as weight, type of food, preferred habitat, etc. These are the properties of the concepts.

- Various types of relationships that can exist between the animals, such as the is-a relationship (e.g. a tiger is a cat), hunt, afraid of, etc. These are the relationships that exist between the concepts.

These definitions form a kind of vocabulary (or language to a certain extent) by which various statements and assertions about the animal kingdom (e.g. “Elephants are herbivores, weigh at least a few hundred pounds, have long noses, and are not afraid of tigers”) can be expressed.

The language in which ontologies are expressed is called an ontology language. The Web Ontology Language, or OWL [2, 42, 76], is one such language. Based closely on Description Logics [3, 4, 65], a family of decidable logical languages that allows domain knowledge to be conceptualized and described in an
object-oriented manner, OWL provides ontology authors with the necessary language constructs to formally define the *vocabulary* for a domain. This vocabulary (or *language* to a certain extent), once defined, can be used to make explicit statements and assertions about the domain. Using the vocabulary defined by the animal kingdom ontology example, one would be able to make assertions, such as the example assertion about elephants above, that represent the knowledge of the domain, and form what is called a *knowledge base*, or KB for short.

Because its vocabulary is well-defined, knowledge bases developed in an ontology language can be read, understood, and reasoned with in an automated fashion using software engines called *reasoners*. Several standardized and efficient implementations of ontology reasoners are available. Among the most well-known ones are FaCT++ [101], Pellet [96], Hermit [55], and Racer [47]. These reasoners allow ontology authors to check their models for any inconsistencies (i.e. satisfiability of model), make automated logical inferences on class-subclass relationships (i.e. subsumption inferencing), and query for object instances satisfying some given descriptions (i.e. instance recognition).

In addition to the automated reasoning benefits above, ontology also offers knowledge-based system designers with two additional important advantages:

- **Intuitive and Familiarity** – Unlike many other logic-based formalisms, ontology employs the familiar object-oriented modeling paradigm (classes, properties, objects, inheritance, etc.). This familiarity allows large and complex bodies of structured knowledge, such as those in biological and medical sciences, to be modeled in a formal yet intuitive and familiar way [39].
• Interoperability and Reusability – With its widespread adoption and strong community support, ontology has become the KR language of choice for many of today’s knowledge-based systems. Due to this popularity, application and domain knowledge that are described in ontology have the highest chance of being understood and reusable by other applications in other domains.

Due to these important benefits, ontology has found practical use in many large projects in e-sciences [61, 15, 75], medicine [51, 71, 80], e-governments [52, 33], business [94, 32], etc, and of course, intelligent systems [59, 105, 23, 38, 103]. In the remaining Sections of this chapter, we will focus our attention to the application of ontology in designing and constructing knowledge-based and knowledge-intensive intelligent systems.

2.2 Computational Planning

Computational planning is the task of coming up with a sequence of actions that will achieve a given set of goals [95]. 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 University of Guelph’s main campus to Toronto’s Pearson airport, board flight AC357 to Paris’ Charles de Gaulle airport, take a cab ride to the KEOD conference in Paris.” Each of these travel steps is called an (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’ 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 combinatorially \[18, 69\]. 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 3.5.3.

From the example above, we can see that simple planning problems can

---

1By simple, we mean planning problems that are deterministic (i.e. all actions have
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 circumstances is an action considered possible? (These preconditions are used by the planner to avoid putting together invalid plans.)

- **Actions’ Effects (aka, Post-Conditions)** – How will each of the work flow actions, when carried out, 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 Advice)** – Advice 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.

### 2.3 Situation Calculus as a logical planning formalism

Originally introduced by John McCarthy as a logical language for axiomatizing dynamically changing worlds, the language of Situation Calculus has deterministic outcomes), instantaneous (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).
been significantly extended by Raymond Reiter as a language for specifying and reasoning about goal based reasoning problems.

There are three fundamental concepts in SitCalc: Action, Situation, and Fluent. We describe these concepts below (This explanation is similar to, but not the same as, the explanation found in [83]):

- **Actions.** Actions represent the various acts that the intelligent agent can carry out in its world. For example, the act of getting a coffee can be represented using the action `getCoffee`. Similarly, drinking coffee, taking a rest, and taking a flight can be represented using the terms `drinkCoffee`, `takeRest`, and `takeFlight(flighNumber)`, respectively. In performing these actions, the intelligent agent affects (i.e. changes) the world in which it lives. It is assumed that this is the only way the agent can influence its surrounding (i.e. all the changes that happen to the agent’s world are the result of its actions alone).

- **Situations.** A situation represents a possible history of the world, and is a first order term constructed from a finite sequence of actions using a special function symbol `do`. For example, the situation `do(drinkCoffee, do(getCoffee, S_0))` is a situation denoting the situation that results after the agent got a coffee and then drank it (`S_0` is a special constant symbol used to represent the initial situation, when the world is thought to begin). Similarly, `do(drinkCoffee, do(getCoffee, do(takeFlight(AC123, S_0))))` denotes the situation that results after someone has taken the flight `AC123`, gotten a coffee and drank it. (Please note that the order at which the
actions appear inside the term is the opposite of the order at which they occur).

- **Fluents.** A fluent is a logical predicate\footnote{SitCalc also recognizes functional fluents, but they are not supported by existing reasoner implementations as of yet.} whose truth value changes from situation to situation. For example, the predicate $HasCoffee(s)$, which represents whether or not the traveler is having a cup of coffee, is a fluent whose value changes from false to true whenever the user gets a coffee, and back to false whenever the user finishes drinking it. Similarly, $IsRested(s)$ is a fluent that represents whether the user is currently rested (i.e. not tired), and $InCity(city,s)$ is a fluent that represents the current location (i.e. city) of the traveler. Intuitively, a fluent can be thought of as the logical counterpart of a state feature, or just feature, in other state-based formalisms.

Using these concepts, intelligent system designers can logically describe the planning problem that underlies their system via a set of logical statements called axioms. We describe these axioms below.

- **Precondition Axioms** Every agent action might have a precondition, which is a set of requirements that need to be satisfied before the action can be performed in the current situation. In SitCalc, this precondition can be described using an axiom of the form

$$Poss(a(x_1, x_2, ..., x_n), s) \equiv \varphi(s)$$

where $Poss$ is a special predicate symbol denoting whether it is possible for the action $a(x_1, x_2, ..., x_n)$ to be executed in the situation $s$, and $\varphi$ is
a SitCalc expression representing the set of pre-condition requirements. For example, the following axiom expresses the fact that it is possible to drink a coffee if one has it:

\[ \text{Poss}(\text{drinkCoffee}, s) \equiv \text{hasCoffee}(s) \]

Similarly, the axiom

\[ \text{Poss}(\text{getCoffee}, s) \equiv \text{True} \]

expresses the fact that it is always possible get a coffee.

In describing a planning problem, the programmer (i.e. axiomatizer) will need to provide one precondition axiom for each agent action.

**Successor State Axioms** All agent actions, when executed, have some effects on the world. That is, they cause the fluents to change their values. One way to describe this dynamic is to specify how each of the fluents will change their values as the result of actions. In SitCalc, this is done using Successor State Axioms, which are logical statements of the form:

\[ F(\bar{x}, do(a, s)) \equiv \Pi_F(\bar{x}, a, s) \lor [ F(\bar{x}, s) \land \neg N_F(\bar{x}, a, s) ] \]

where \( F \) is the fluent symbol, \( \Pi_F \) is a formula representing the logical condition that causes the fluent to be true, and \( N_F \) is a formula representing the logical condition that causes the fluent to be false). For example, to specify how the \( \text{HasCoffee}() \) fluent would respond to various agent actions, the following axiom can be used:
\[ HasCoffee(do(a, s)) \equiv a = \text{getCoffee} \lor [ HasCoffee(s) \land \neg a = \text{drinkCoffee} ] \]

This axiom completely specifies how the value of the HasCoffee fluent would respond to all possible actions being performed. In particular, it states that HasCoffee is true if the user just performed a getCoffee, and will remain unchanged (i.e. keep its previous value) as long as the user didn’t drink it.

In describing a planning problem, the programmer (i.e. axiomatizer) will need to provide one successor state axiom for each fluent.

From a declarative standpoint, the above concepts and axioms provide planning system designers with enough expressiveness to completely specify or describe a planning problem. Using the actions and fluents concepts, planning system designers can enumerate and describe all the possible agent actions, as well as all the different features (i.e. fluents) of the world. Using precondition and successor state axioms, they can also describe the actions’ pre-conditions, as well as their effects on the world (via describing the dynamic of each fluent). These descriptions, as a knowledge base, form a complete description of the planning problem.

While this description is useful for the purpose of specifying the planning problem in a precise and logical way, it is often not very useful for finding the solution to the problem – most planning problems of practical significance are
often too complex for a logical reasoner (i.e. planner) to solve declaratively. To help overcome this issue, SitCalc also provides planning system designers with the ability to encode procedural advice on how the planning problem can best be solved. These procedural advices can take one of the following forms:

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>$\delta_1;\delta_2$</td>
<td>Subprogram $\delta_1$ should be executed before subprogram $\delta_2$</td>
</tr>
<tr>
<td>Conditional</td>
<td>if $\phi$ then $\delta_1$ else $\delta_2$</td>
<td>Execute $\delta_1$ if $\phi$ is true, and $\delta_2$ if otherwise</td>
</tr>
<tr>
<td>Loop</td>
<td>while $\phi$ do $\delta$</td>
<td>Execute $\delta$ until $\phi$ is no longer true</td>
</tr>
<tr>
<td>(Smart) Choice</td>
<td>$\delta_1</td>
<td>\delta_2$</td>
</tr>
<tr>
<td>Procedure</td>
<td>proc$(p,\delta)$</td>
<td>Program expression $\delta$ can be executed by calling procedure $p$</td>
</tr>
</tbody>
</table>

Continuing with our intelligent travel advisor example, assume that we want our travel advisor to ask its users to either take a rest or have a coffee right after a (tiresome) flight. To accomplish this, we can provide the system with the following procedural advice:

```
procedure (  
    TravelByAir(flightNo),
    takeFlight(flightNo); (takeRest | getCoffee ; drinkCoffee)
)
```

As shown in Table 2.1 above, the semicolon represents the sequential operator, while the vertical bar represents a choice. This advice, therefore, signals the SitCalc reasoner that, in its search for a valid travel plan, when-
ever it encounters a call to \textit{TravelByAir}(flightNo), it should take the action \textit{takeFlight}(flightNo), followed by a choice between taking a rest and having a coffee, whichever is possible.

Once the system designer has completed the declarative description of the underlying planning problem, he or she can give this description, together with any procedural advices, to a SitCalc reasoning engine, such as the GOLOG [68] engine, and ask it to compute the solution to his or her problem.

Once the planning problem has been described declaratively in a knowledge base, and all the procedural advices (which are optional) have been encoded, possible solutions (i.e. plans) to the problem can be found by invoking a SitCalc reasoning engine. One standard implementation of such engine is the GOLOG interpreter [68]. This interpreter is implemented in Prolog, and takes as its inputs the declarative knowledge base, as well as any procedural advices, and produces a situation term representing the specific sequence of actions that would accomplish the goal, taking into account all the action precondition and effects. This sequence is the solution the user of the planning system is looking for.

Great wealth of details about the language of SitCalc and Golog can be found in Reiter’s book, “Knowledge In Action” [93], and Brachman and Lesveque’s book, “Knowledge Representation and Reasoning” [13].

2.4 Planning in Ontology-Driven Applications

As we have discussed in Section 2.1, Ontology offers knowledge-based intelligent application designers with several important advantages as a knowledge
representation language. First, it provides the system designers with a formal (i.e., logic-based) yet intuitive (i.e., a familiar object-oriented model) formalism in which to model their application knowledge. This formality, coupled with the availability of standardized and efficient reasoner implementations, allows application knowledge to be modeled in a precise (i.e., machine interpretable) and modular (i.e., incremental updating of model) way \[39\]. Second, its widespread adoption affords system designers with the standardized medium needed to capture their application knowledge in an inter-operable (i.e., application and domain independent) and hence reusable way \[45\].

Despite these clear and important advantages, ontology as a KR formalism still remains a difficult choice for many knowledge-based intelligent system designers – while they provide efficient support for ontological reasoning, ontology languages lack the native support for planning \[34\] \[28\] which, as we explained in Section 2.2 above, is a highly desirable capability for many intelligent systems. In this section, we explain this important limitation, and survey various existing approaches to bringing planning support into ontology-driven applications. We start out with an explanation of the fundamental differences between ontological reasoning and planning. Next, we survey existing approaches in bringing planning support to ontology-driven applications, and discuss their promises, advantages and disadvantages. Lastly, we discuss a list of features that we considered to be desirable for an integration framework that aims to bring planning capabilities into ontology-driven applications.
2.4.1 Planning vs Ontological Reasoning

Despite their apparent similarities, planning and ontological reasoning are actually two different types of logical reasoning. In ontological reasoning, the reasoner’s primary task is to infer new structural knowledge from a given knowledge base (i.e., the asserted knowledge). That is, the reasoner uses the logical definitions of the concepts and objects in the model to make inferences about their taxonomic (i.e., “class-subclass” and “instance-of”) relationships [50]. In planning, on the other hand, the reasoner’s primary task is to come up with a valid plan that could accomplish a given set of goals or objectives. That is, the reasoning engine uses the logical descriptions of the available actions, together with their preconditions and effects on the environment, to piece together a concrete sequence of actions that would cause the world to transform from its current state to the desired goal state. Also, in ontological reasoning, the environment is considered to remain static for the entire duration of the reasoning process (that is, no state change occurs during the reasoning process), whereas in planning, the environment is thought to have changed its state with every action considered.

2.4.2 Existing works on integrating planning into ontology-driven applications

Because many intelligent systems derive their purposeful decision-making capability via a goal-based mechanism, there has been a lot of interest in finding a practical way to incorporate planning capability into ontology, and a lot of work has been reported in this area. On a high level, these works can be divided
into three main approaches: Language-based, Parallel Modeling, 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 is described in ontologies, while planning-related information is described separately in a rule-based language. In the third approach, planning-related knowledge is described using an ontology, along with other application knowledge, and translated into an executable rule-based planning program. We describe these approaches in more details below.

Language-based approaches

In these approaches, the underlying language (i.e., Description Logics) is modified, extended, or restricted to support planning. The primary rationale behind this approach is that since planning has been well supported by rule-based formalisms such as Horn Logic, one could potentially integrate planning capabilities into Ontology by incorporating language features from these formalisms.

Two of the most well-known examples of this language-based approach are the Semantic Web Rules Language (SWRL) [58, 48] and the Description Logic Programs (DLP) [64]. Generally speaking, SWRL is an extension to Description Logics, while DLP is a restriction of Description Logics [64]. In SWRL, the primary goal is to extend Description Logic’s expressiveness to give ontology users the ability to describe their knowledge using rule-like syntax (in addition to the usually Description Logics syntax). In its unrestricted form, SWRL can be considered a union of DL and rules (i.e, Horn Logic [63]). This language, while very expressive, is an undecidable [53], and therefore is not very useful from an automated reasoning point of view. If automated reasoning
is important, the users can try a restricted version of SWRL, called DL-Safe SWRL, which is decidable and supported by reasoner implementations such as KAON [12]. One of the primary limitations of SWRL as a rule language is that, like Description Logics, it does not allow $n$-ary predicates, where $n$ is greater than 2 (only unary and binary predicates are allowed in the language). DLP, on the other hand, can be considered to be the intersection between Description and Horn Logics. The rationale behind this language is that, because the full expressive power of Description Logics is not needed for the majority of existing ontologies [44], if one is willing to stay within the rule fragment of Description Logics (i.e., the intersection between DL and HL), rule-based reasoning can be guaranteed. It is often argued [53] that DLP is too restrictive as a language, however. As well, because it is a proper subset of DL, DLP also allows only unary and binary predicates.

Generally speaking, the main advantage of language-based approaches such as SWRL and DLP 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 these approaches, however, is that they are inherently difficult, and success has been very limited so far [53, 79, 5]. In addition to the semantic (i.e., open world vs closed world[2]) and complexity (i.e., decidable reasoning algorithms) issues, 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 rea-

---

[2] Description Logics assume an open-world model, in which all unasserted facts remain unknown, while Horn Logic assumes a closed-world model, in which all unasserted facts are automatically assumed to be false. 
soner 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.

Other efforts in combining rules and ontology can be found in [57, 41, 37].

Parallel Modeling approaches

Another popular approach to bringing planning to ontology-driven applications is the “parallel” approach in which planning and application knowledge is modeled separately in two parallel worlds: planning-related knowledge is described in a (rule-based) planning language, while other application knowledge is 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.

Within the ontology-driven work flow composition community, for example, several works have been reported to follow this approach. [8], [106] and [31], for example, describe 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 algo-
rithms, together with other relevant information such as their pre-conditions,
etc., and then compose the workflows by putting these algorithms together.

As compared to the language-based approaches, parallel modeling has the
advantage of practicality – existing tools and technologies can be employed
to solve ontology-driven planning problems without having to wait for new
languages or frameworks. There is one important disadvantage in these ap-
proaches, however. Because planning-related knowledge was encoded in a
planning language instead of ontology, reusability and interoperability of this
knowledge is therefore reduced. To some degree, this weakness defeats the
purpose and benefits promised by an ontology-driven approach.

**Translation approaches**

Another promising approach to bridging planning and ontological modeling
is to 1) use a dedicated ontology to describe all planning-related information,
and then 2) use an automated translation engine to translate this information
into an executable planning program that can be executed in a rule-based
framework. This approach is represented by the works reported in [40] and
[90].

In [40], the authors describe a dedicated planning ontology, called
PLANET, which provides various language constructs for describing military
action plans:

- Objectives, sub-objectives, and objectives decomposition (i.e., how main
  and sub goals are related in a hierarchy). E.g., “Eliminate enemy threats
to US allies by either a) Destroy all SSM launchers and launch sites by day 5 or, b) Disrupt and disable enemy C2 infrastructures by day 2.

- Tasks, task constraints, and tasks-accomplish-objectives relationships. Eg. “A mechanized division attack the northern flank to seize X to protect Y”.

- Decision points, Ordering of tasks (A to be finished before B), Temporal constraints (Complete A by day 5).

One of the main limitations of PLANET, from an ontology-driven intelligent system point of view, is that while the list of language constructs it provides is comprehensive and expressive for the purpose of describing complex plans, its main intent was to describe plans as opposed to planning problems (the first is a solution of the second). As such, PLANET lacks the necessary constructs (preconditions, effects, world’s dynamics, cost/reward, etc) needed to drive a planner, and therefore is inadequate for the purpose of building a ontology-driven planning systems. Similar course of action ontologies have also been described in [27, 26].

In [90], the authors presented a generic planning ontology that could potentially be used 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. The main limitation of this approach is that, while it has made a good case for the potential ben-

3These might sound like tasks or actions but they can also be thought of as verb-based objectives.
efits of describing planning problems using a dedicated planning ontology, it stopped short of demonstrating if and how the planning problems, once described, can be translated and executed in a planning engine to produce the desired results. In order to do so, a few important practicality issues are to be expected. The first issue is that of translatability: How does one ensure that all planning problem descriptions are translatable into equivalent rule-based programs? Because ontology (i.e., Description Logics) and rules are two different languages, there is the possibility that an ontological description cannot be translated into an equivalent rule-based version. The second issue 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 2.2, real-world problems often result into planning problems that are beyond even the most capable planners, and an effective planning framework needs to provide support for coping with this complexity.

In the next chapter, we report our efforts in pursuing this idea further by addressing these practicality issues, and present a complete translation-based, ontology-driven planning framework for ontology-driven applications.

2.5 Web Service Composition

Another area of research in which computational planning and ontology are often considered and employed together is the area of service-oriented computing [1,83]. In SOC, software is designed according to an architecture that allows independent units of computational functionality, known as web services, to
be combined together at deployment time to make larger functionalities and applications. Using this architecture, loose coupling between the individual web services is achieved by having the web services communicate with one another via messages in well-defined protocols (e.g. SOAP [11, 23]) that are based on existing web technologies (e.g. HTTP [7] and XML [14]).

One of the major research interests in service-oriented computing is the issue of web service composition, which seeks to answer the question of how to make it’s easier and more effective for system designers to find, select, and integrate existing web services into applications that can fulfill a given set of requirements. Given the heterogeneous, distributed and decentralized nature of web services, web service composition is a challenging problem, and a lot of efforts have been reported in this area [98, 78, 17, 91, 77, 19, 82, 66, 6, 102, 66].

From a high level perspective, a web service composition task can be thought of as two related but independent sub-tasks:

- **Synthesis** – Given a set of existing individual web services, the task of coming up with a proper compositional structure that combines these services together to fulfill a particular set of requirements is often referred to as a synthesis task. This task often requires careful considerations of the individual web services’ requirements and effects, and is often treated as a computational planning problem: If one considers the individual web services as different actions that can be performed by an imaginary agent, then the desired composition can be considered a plan, and synthesis can be considered as a planning task.
• Orchestration – This is the task of managing the proper execution an already-synthesized web service compositional structure.

Several languages have been proposed to facilitate the orchestration of web services. Most notable among these are the Web Service Flow Language (WSFL) from IBM [21, 20], XLANG from Microsoft [100], and BPEL4WS from a consortium of software vendors [104, 60]. These languages, commonly referred to as flow or execution languages, allow a composition of web serviced to be described in such a way that its execution can be carried out, monitored and managed by an orchestration framework.

From a high level viewpoint, these languages bear some resemblances to the language supported by our planning ontology (which is explained in Chapter 3), as they also contains language constructs for describing the work flow of a composition. On a closer inspection, however, the reader will notice that these languages were mainly designed to handle orchestration instead of synthesis. As such, they are not particularly well-suited for the purpose of doing computational planning. From a computational planning perspective, the flow languages are more suited for the purpose of describing plans as opposed to planning problems (The former can be thought of as a solution to the later – One often needs to “solve” a planning problem to find a plan).

A lot of work has also been reported on the synthesis of web service compositional structures. Because synthesis lends itself well to a planning solution, many of these works employ planning as the primary mechanism to automatically come up with a compositional structure [21, 49, 99, 89, 62, 107, 88, 73]. Generally speaking, planning-based approaches to web service composition/synthesis can be divided into two separate groups: those that operate
at a syntactic level; and those that operate at a semantic level. Approaches that operate at the syntactic level typically make use of the WSDL descriptions \[24\] of the individual web services to produce input-compatible compositional services. Because they do not need to deal with semantic and ontology-based information, these approaches are less relevant to our approach.

Approaches that operate at the semantic level, on the other hand, typically make use of the semantic descriptions of the individual web services in order to produce the desired composition. This semantic information, which is often encoded in an ontology language such as OWL-S \[72\], might contain the pre and post conditions of the web services, and is often translated into a planning problem in a planning language such as PDDL \[74, 36\]. Figure 2.1 shows an example of such approaches.

![Figure 2.1: A recent example \[50\] of a (semantic) web service composition approach that operates by translating ontology-based web service descriptions into a planning problem in a planning language.](#)
At a high level point of view, these translational approaches to semantic web service composition are very similar to our proposed framework. First, they both deal with the issue of employing computational planning on ontologically represented knowledge. Second, they both employ translation to bring ontologically represented knowledge into a planning language where the limitations of ontological reasoning can be overcome. On closer inspection, however, these approaches differ from ours in two important and fundamental ways. First, instead of mixing web services descriptive information with planning-related knowledge (i.e. the pre and post conditions) in the same ontology, our proposed framework encourages separation of concerns and allow domain knowledge to be modeled separately and independently from planning knowledge. This separation is shown in Figure 3.4 by the clear separations between the domain ontologies and the planning ontology, as well as the domain knowledge bases and the planning knowledge base. As discussed in Section 3.1, this separation of concerns allow domain knowledge to be modeled in an application/purpose independent, and hence more reusable, way.

The second difference between our framework and existing semantic web service composition approaches is much more fundamental: OWL-S is an OWL-based ontology that was designed specifically for the purpose of web service composition. Composition approaches that are based on OWL-S are, therefore, purpose-specific, and might not apply well to other planning problems outside the domain of web service composition. Our planning ontology, on the other hand, is designed based on Situation Calculus, which is a generalized action formalism designed for axiomatizing dynamically changing worlds. This action formalism provides system designers with the capability to model highly
complex dynamical worlds that might exhibit features such as stochastic actions [10, 35], concurrency [30], action duration and temporal constraints [92], exogenous actions and sensing [29], complex cost/reward structures [10, 9], etc., and offers a much more general route toward the integration of computational planning into ontology-driven applications.

This concludes our background and related works chapter. In the next chapter, we propose a new translational approach to integrating computational planning into ontology-driven applications, along with the rationales for such an approach.
Chapter 3

A Practical Planning Integration Framework for Ontology-Driven Applications

In this chapter, we describe our proposal for an alternative approach to supporting planning in ontology-driven applications. We start out by describing the overall approach, and discuss why such an approach would be more practical and reusable than existing approaches. Then, we describe the major challenges that need to be overcome, together with the observations and intuitions that allowed us to overcome these challenges. Then we move on to describe the framework design, its operations, and illustrate its usage using an example ontology-driven planning problem.
3.1 Rationales for a Translation Approach

As we have discussed in Chapters 1 and 2, existing approaches to incorporating planning reasoning into ontology-driven applications still have important limitations. Most language-based approaches to planning reasoning in ontology-driven applications are still in their early stages. Parallel modeling approaches, on the other hand, have the disadvantage of reusability. In the pursuit of a more practical and reusable approach to integrating planning reasoning into ontology-driven applications, we propose an alternative, translation-based approach that is illustrates in Figure 3.1.

Figure 3.1: A translation approach to incorporating planning into ontology-driven applications. This architecture will be further developed in Figure 3.4.

Using our proposed approach, the ontology-driven application developer would:
1. Describe his or her planning reasoning problem in an ontology, just as
   he or she would with other application knowledge, and

2. Have our framework transparently translate this problem description into
   an executable rule-based program, execute it in a rule-based reasoning
   framework, and return the result to him or her.

More specifically, the ontology-driven application designer would use the
pre-defined ontological constructs that are provided by our Planning Ontology
to describe his or her planning problems into a knowledge base. The framework
will translate this problem description into an equivalent executable rule-based
planning program, execute it in a underlying planning framework, and return
the result back to the user.

If successful, our translation-based approach would offer several important
advantages over existing approaches. As compared to language modification
approaches, our approach offers the advantage of practicality and “safety”.
Practical because our approach allows planning reasoning to be integrated
into ontology-driven applications using existing tools and mature formalisms/
languages. Safe because new language extensions not only often take time,
but also carry the risk of failing to become a mainstream choice down the
road. As compared to parallel modeling approaches, our approach offers the
advantage of reusability and interoperability – Because planning problem
descriptions are expressed in ontology, they are guaranteed to be understood,
and hence reusable, by other ontology-aware applications, including those that
might become available in the future.

More specifically, our proposed framework and its architecture offer four
main advantages as compared to existing approaches:

- *Separation of Concerns* – Because planning-related knowledge are kept separated from the domain ontologies and KBs, these KBs can be developed and maintained independently from the planning application itself. This independence not only makes the domain ontology simpler to develop and maintain, but also makes it application and purpose-independent, and therefore much more reusable for future applications.

- *Framework Independence* – 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.

- *Mature and Proven Formalisms* – 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 planning reasoning capability in ontology-driven applications.

- *Transparency and Non-Intrusiveness* – Using our framework, the ontology-driven application designer can continue to think and work in the ontological modeling 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 describe his planning problem in his familiar ontology editing environment, and have the framework handles all the mappings for him.

3.2 Addressing the Primary Challenges

As we have discussed in Chapter 1, realizing the benefits of the proposed translation approach requires answering several important questions. In this section, we repeat these questions and explain how we will address the challenges that underlie these questions.

3.2.1 Representability

Representability – Is it possible to describe or model planning reasoning problems in ontology?

Traditionally, planning reasoning problems are modeled in rule-based languages such as Horn Logic (HL) [63]. As shown in Figure 1.1, this language is not a proper subset of Description Logic, and so a probability exist for some planning knowledge that were expressible in HL become non-expressible in DL/ontology. Also, while HL assumes a close-world model, DL assumes an open-world model. This semantic mismatch can also cause some expressibility issues.

In this section, we describe how we are able to overcome the representability challenge mentioned above using a rather simple but crucial observation. We begin by describing an important difference between Description Logics (DL)
and rule-based languages such as Horn Logic, and explain why this difference makes describing planning reasoning problems in ontology a difficult task. Then we describe our observation and explain how it allows us to overcome this difficulty and describe planning reasoning problems in ontology.

As we have discussed in Chapter 2, planning reasoning problems are traditionally described as executable programs in rule-based formalisms such as Horn Logics (i.e. Prolog). Our proposed approach, on the other hand, requires that these problems descriptions are made in ontologies instead. Due to an important difference between Description Logics (DL) and Horn Logic (HL), the attainability of this requirement is not a given. Consider, for example, the following Horn clause:

\[
\text{Triangle}(p1,p2,p3) \leftarrow \text{Point}(p1) \land \text{Point}(p2) \land \text{Point}(p3) \land \\
\neg (p1 = p2) \land \neg (p2 = p3) \land \neg (p3 = p1)
\]

This statement, which is often referred to as a rule, expresses a rather simple fact that any set of three distinct points makes a triangle. Expressing this fact in ontology (i.e. Description Logics) is not a straight forward task however. The primary reason of this difficulty is that DL only allow unary and binary predicates (i.e. predicates that has 1 or 2 “arguments”) in its language. Because \text{Triangle} is a tertiary predicate (i.e. it has 3 arguments), the above statement cannot be directly expressed in DL.

One common technique ontology modelers often use in order to get around this restriction is to replace the \(n\)-ary predicates (predicates with \(n\) arguments, where \(n\) is bigger than 2) with an equivalent set of binary predicates. The above
statement, for example, can be modeled using an equivalent set consisting of three statements involving the binary predicate such as \textit{TriangleVertice}. While this technique often works for the modeling of object-oriented worlds (because classes and their relationships are well described using unary and binary relations), it doesn’t work very well for the modeling of planning reasoning problems. As will become apparent in Chapter 4 below, modeling real-world planning reasoning problems will often involve the use of many complex logical expressions that contain several n-ary predicates which can not be easily replaced by an equivalent set of binary relations. As such, this relation “decomposition” technique is not always elegant, intuitive or even possible.

To overcome this challenge, we make a rather simple yet important observation: With a proper set of ontological constructs to represent logical operators and concepts, one can represent (arbitrary) Horn clauses in Description Logics. To see how this observation works, imagine that we have the following set of concepts:

- \textit{Conjunction} – A concept representing a logical conjunction of two or more logical expressions.
- \textit{Disjunction} – A concept representing a logical disjunction between two or more expressions.
- \textit{Negation} – A concept representing the logical negation.
- \textit{Relation} – A logical concept representing a logical relation.
- \textit{Implication} – A concept representing a logical implication.
- \textit{Equality} – A concept representing the special relation of equality.
Using these ontological concepts, one can describe the example rule above as shown in Figure 3.2. As can be seen in the figure, a Horn clause (i.e. rule) can be represented using an object of type Implication, which contains two attributes, Head and Tail. In this example, the Head attribute is set to an object of type Relation whose name is “Triangle” and arity is 3. Similarly, the
The Tail attribute is set to a *Conjunction* between 6 logical expressions.

As the above example illustrated, while ontology does not provide support for planning reasoning, it is fully capable of describing Horn clauses, and hence planning reasoning problems. This observation – that an appropriate set of ontological constructs can be used to allow statements that are outside the expressibility boundary of ontology to be described – will serve as a key for us to overcome the representability challenge.

### 3.2.2 Translatability and Constrainability

*Translatability and Constrainability* – Assuming that a planning reasoning problem has been successfully described/modelled in an ontology, is it always possible to translate it into equivalent executable rule-based programs?

As we have seen in Figure 1.1, Description Logic is also a non-proper subset of Horn Logic and, in order for our proposed approach to be feasible, it is extremely important that we can find an effective way to ensure the users do not describe their reasoning problems in such a way that cannot be translated into the language of Horn Logic.

In this section, we make another simple yet important observation that will guide us to a solution to this challenge: A well-defined ontology can be thought of as a language – The concepts in the ontology constitute the vocabulary of the language, while the roles in the ontology dictate the ways in which the terms in the vocabulary can be combined together to form statements. Furthermore the expressiveness of this language can be controlled through the concepts and roles that are contained in the ontology.

We illustrate this observation by considering the following two example
implications:

\[
\text{GradStudent}(x) \leftarrow \text{Student}(x) \land \text{TakeCourse}(x, y) \land \text{GraduateCourse}(y)
\]

\[
\text{MastersStudent}(x) \lor \text{PhDStudent}(x) \leftarrow \text{Student}(x) \land \text{TakeCourse}(x, y) \land \text{GraduateCourse}(y)
\]

The first one is a valid Horn clause, whereas the second one is not, because it contains a disjunction in the conclusion (which violates the so-called “No disjunction in the head” rule). Given all the relevant concepts (i.e. \text{Student}, \text{GradStudent}, \text{MastersStudent}, \text{PhDStudent}, \text{Course}, etc.) and roles (i.e. \text{TakeCourse}), the user can express both types of statement using the \text{Implication}, \text{Disjunction} and \text{Conjunction} concepts from our previous example. Assuming that we want the user to make statements of the first type only, we can simply restrict the domain of the \text{Implication}’s \text{Head} attribute to just the class of \text{Relation}. (That is, we restrict the \text{Head} attribute to accept only objects of type \text{Relation}, instead of any object representing arbitrary logical expressions). This restriction ensures that the user never produce any implication expression that contain a disjunction in the head. Of course, without the necessary concepts (i.e. \text{GradStudent}, \text{GradCourse}) and roles (i.e. \text{TakeCourse}), the user would not be able to make those statements in the first place. As such, the vocabulary itself is also a source of control. By simply adding or removing concepts and roles from the ontology, one can control the types of expressions the user would make.

With the above examples in mind, one could see that by being selective and
careful with the language constructs provided by the Planning Ontology, we can implement an effective yet transparent and non-intrusive way to restraint the user from making descriptions that is non-translatable to Horn Logic. This intuition is illustrated in Figure 3.3 below.

![Figure 3.3](image)

Figure 3.3: *The Planning Ontology acts as a restrainer that helps ensure the description of the planning reasoning problem always falls into a sub area of Description Logics that is translatable to an executable program in Horn Logic.*

This observation – that an ontology can be used as a language, and that it can be used to transparently and non-intrusively control the type of expressions the user can make through it – provides us with the much needed assurance that any potential translatability issues can therefore be overcome. In Section 3.7.1 below, we will revisit this discussion and describe in details how we turned this intuition into a technique that systematically ensure that all planning problem descriptions made by the users are translatable to Horn expressions.
3.2.3 Effectiveness and Reusability

*If such a framework is feasible, will it be effective enough to handle non-trivial, real-world planning reasoning problems? Also, can it be re-used across application domains?*

To be a practical alternative to existing approaches, such a framework would need to be effective enough to handle real-world planning reasoning problems. Also, to demonstrate a clear advantage over ad hoc approaches, this integration framework would need to be generic enough to be reusable across multiple application domains.

We discuss how we addressed these practicality requirements with the case studies in Chapter 4.

3.3 Thesis Statement

Armed with the above observations and intuitions, we set out to prove the feasibility of our proposed translation approach with the following thesis statement:

**Thesis Statement** – *Planning in ontology-driven applications can be effectively supported via the use of a reusable translation integration framework.*

There are three major components to this claim – Feasibility; Effectiveness; and Reusability. With feasibility, we answer the question of whether or not it is possible to overcome the first two challenges – namely, representability and translatability – and build an integration framework. With effectiveness, we answer the question of whether or not the resulting framework is effective
(i.e., powerful and flexible) enough to handle complex, real world goal-based reasoning problems. With reusability, we answer the question of whether or not the resulting framework can be reused across applications and domains without requiring extensive modifications.

In the next chapter (Chapter 4), we will explain in details how each of these components will be demonstrated. In the remaining parts of this chapter however, we will proceed to describe our approach in details.

3.4 Proposed Architecture

In this section, we describe a general architecture that ontology-driven application developers can use to build intelligent applications using our proposed translation approach. Figure 3.4 illustrates this architecture.

As shown in the figure, this architecture involves the use of several ontologies:

- **Planning Ontology** – This ontology contains pre-defined concepts by which the application developer can describe the planning reasoning problems that underlie his or her intelligent application. This ontology is supplied as part of our reusable framework, and does not need to be modified on a per-application basis (i.e. the language constructs in this ontology is generic to all planning applications).

- **Planning KB** – This is the knowledge base in which the application developer actually describe, using the language constructs provided by the Planning Ontology, the planning reasoning problem that underlies
Figure 3.4: A generic architecture for building modular and reusable planning ontology-driven applications. The Planning Ontology (described in Section 3.5) and the Translation Engine (described in Section 3.6) are supplied by us as part of our reusable framework. The Goal Ontology, Domain Ontology, Domain KB and Planning KB are supplied by the ontology-driven intelligent application developer on a per-application basis. The Goal Statement is supplied by the end user on a per-query basis.

the particular application that he or she is building. This KB contains all the planning-related knowledge that are needed to drive the planner and build the plans. Unlike the Planning Ontology, the Planning KB needs to be individually created for each individual application, as the knowledge is captures is specific to the individual application.

• Domain Ontology and Domain KB – The Domain Ontology and its associated knowledge base, the Domain KB, are where all the domain knowledge are to be captured. They contains all the relevant knowledge about
the domain in which the application operates. For the example trip planning application described in Section 2.2 this KB would 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. Ideally, this ontology and knowledge base should describe domain knowledge in an application and purpose independent way, and hence can be reused between all applications operating in the same domain. This is one of the main benefits of our proposed architecture – Application-specific planning knowledge can easily kept separated from application-independent domain knowledge, making the later much more reusable.

- **Objective Ontology and Objective Statement** – An objective statement is a planning request that the application user submits as input to the application every time he or she wants it to produce a plan. For the intelligent trip planner example described in Section 2, these objective statements could look something like “Find a way to get from University of Guelph to the KEOD2011 conference center” or “Find a way to get from Paris to Nantes without flying”, etc. As such, the Objective Ontology is there to provide the user with the necessary language constructs to express his or her objectives in a concise way. This ontology is most likely to be application specific, and hence needs to be provided by the application developer on an application by application basis.

In the next chapter (Chapter 4), we will show details examples for each of these ontologies and KBs through the use of two different case studies. In the
remaining parts of this chapter, we will focus our discussion on the Planning Ontology and the Integration Framework.

3.5 The Planning Ontology: Pre-Defined Constructs for Describing Planning Problems

In this section, we focus our attention to the Planning Ontology, shown as the dark blue oval on the top right corner of Figure 3.4 and discuss in details the various language constructs (i.e. concepts and roles) that it offers.

As we have mentioned earlier, the Planning Ontology provides the application developer with the necessary vocabulary (i.e. language) by which to describe the planning problem that underlies his or her application. To allow this ontological description of the planning problem to be as generic (and hence reusable) as possible, it is important to ensure that the vocabulary (and language) that is provided by the Planning Ontology is independent of any specific planning framework and/or their implementation. To satisfy this requirement, we designed our Planning Ontology based on the language of Situation Calculus (SitCalc) – The implementation-independent logical language for axiomatizing dynamically changing worlds that we have reviewed in Section 2.3. As such, many of the concepts in the Planning Ontology are modeled after the concepts of the SitCalc language, and we will point out the relationships between the two set of concepts in our discussion.

As an overview, the constructs in the Planning Ontology can be divided into three main groups:
• **Logical Constructs** – This group consists of concepts that represent standard logical operators, such as conjunction, disjunction, negation, etc. These concepts, while technically not directly related to planning, will allow planning concepts to be logically described in a convenient and translatable way.

• **Basic Planning Constructs** – This group consists of concepts that are needed to describe the basic planning ingredients mentioned in Section 2.2. They provide the ontology-driven application developer with a necessary vocabulary to declaratively describe his or her planning problem.

• **Planning Heuristic Constructs** – Real-world planning problems are often too complex for a planner to solve exhaustively. To give the application developer more control over the planning process, the Planning Ontology provides the developer with an additional set of language constructs that he or she can use to provide heuristic Advice to the planner. This group of constructs allows procedural Advice to be expressed as partial programs that help make the planning problem more computationally effective and tractable.

Below, we will continue using the Trip Planning example discussed in Section 2.2 to illustrate the various language constructs provided by the Planning Ontology.

### 3.5.1 Logical Constructs

The first group of language constructs provided by the Planning Ontology are those that are needed for describing logical expressions. Figure 3.5 shows these
constructs.

As can be seen in the figure, *LogicalExpression* is the top-level concept from which all other logical concepts are defined. That is, a *Conjunction* is a *LogicalExpression*. So do *Disjunction* and *Negation*, as well as the *Constraint* concepts. Also, while the *Conjunction*, *Disjunction* and *Negation* concepts are recursive and can contain other logical concepts (including themselves) as a sub-expression, the *Constraint* concepts are not recursive and can not contain other logical concepts.

We have seen an example of how these logical constructs are used in Figure 3.2 and we will see many more *LogicalExpression* examples in this section.
3.5.2 Basic Planning Constructs

The second group of language constructs in the Planning Ontology are those that are needed for describing the basic ingredients of the planning problem. Figure 3.6 shows these constructs and their relationships.

Figure 3.6: Basic planning constructs of the Planning Ontology. These pre-defined constructs constitute a pre-defined vocabulary from which planning problems can be described in a framework and planner independent way.

Actions

The \textit{Action} concept is used to describe planning actions. To describe an action, the application developer would instantiate an Action object and populate it with appropriate values:

- The name of the object is populated with the action’s name.
- The \textit{Arguments} property is set to an array of variables representing the action’s arguments.
- The \textit{PreCondition} property is set to an object of type \textit{PreCondition}, described below, representing the action’s precondition.
Figure 4.3 shows how the Action concept is used to describe the various actions from the Trip Planning problem. As can be seen there, the TakeFlight, TakeTrain, and TakeBus actions each takes a single arguments (i.e. parametrized), whereas the GetCoffee, DrinkCoffee and TakeRest are parameter-less. More detailed and complex examples on the usage of the Action concept can also be found in Section 4.2.4.

Action’s Pre-Conditions

The PreCondition concept is used to describe action’s preconditions. To describe the precondition of an action, the application developer would instantiate a PreCondition object and populate its LogicalExpression property with an object of type LogicalExpression representing the logical condition under which the action can be selected.

In Figure 4.7 we can see that the precondition for the action TakeFlight, for example, is that the user is located in the same city as the flight’s origin. More detailed and complex examples on the usage of the PreCondition concept will be discussed in Section 4.2.4.

State Features

As we have discussed in Section 2.3, the environment’s dynamics (i.e. how it responses to the agent’s actions being performed) can be described using a set of logical predicates, each representing one particular feature (i.e. aspects) of the environment’s state. As actions are performed against the environment, the state features change their values to reflect the new states that result from these actions. Not all actions will cause all state features to change.
their values, however. Each feature, in fact, only responds to a certain conditions, and remains the same in all other circumstances. As an example, the state feature representing the user’s current location is only affected by the actions TakeFlight, TakeBus or TakeTrain. All other actions (i.e. TakeRest, GetCoffee and DrinkCoffee, or any other actions had they existed) do not effect the value of this feature.

In our framework, state features can be described using the StateFeature concept provided by the Planning Ontology. In particular, to describe a state feature, the application developer would instantiate a StateFeature object and populate it with appropriate property values:

- The name of the object is set to the feature’s name
- The Arguments property is set to an array of variables representing the feature’s arguments.
- The DefiniteValueCond property is set to a LogicalExpression object representing the logical condition in which the feature will take on a definite value.
- The InvariantCond property is set to a LogicalExpression object representing the logical condition in which the feature will remain unchanged (invariant).

Figure 4.6 shows an example in which the StateFeature concept, and its property values, are used to describe the dynamic of InCity, the state feature representing the user's current location. In that example, InCity is shown to change its value whenever the user take a flight, train or bus, and remain
invariant if neither of those actions are taken. In Section 4.2.4 we will revisit the StateFeature concept and provide several more examples on the usage of the StateFeature concept.

Readers who are familiar with SitCalc and Golog will recognize that the StateFeature concept is the equivalence of the Fluent in SitCalc, while the InvariantCondition and DefiniteValueCondition represent the negative and positive clauses of Reiter’s Successor State Axiom, respectively.

GroundFacts

The GroundFact concept and its sub-concepts are used for defining new logical relations using data from existing concepts. A ground fact can be thought of as an instruction to the underlying framework on how to construct a new logical relation using the data from an existing concept from the domain ontology. We will explain these concepts in details using several illustrative examples in Sections 4.1.4 and 4.2.4.

3.5.3 Planning Heuristics Constructs

The third set of language constructs provided by the Planning Ontology are those that can be used by the application developer to provide heuristic Advice to the planner and help it to come up with a plan faster.

In Section 2.2 we observed that real-world planning problems are often too complex for even the most advanced planners to solve exhaustively. We also observed that any practical planning framework, in order to cope with this complexity in an effective way, must therefore provide the application
developer with an effective and convenient mechanism for providing heuristic advice to the planner and helping it cut down on the search. In Section 2.3, we reviewed one such mechanism – The Partial Programming mechanism [93] supported by the SitCalc language. As a quick refresher, a partial program is essentially a generic plan that has not been completely fleshed out yet (hence the name partial). “First, take the Greyhound 123 bus from Guelph to Toronto. Then, either take a direct flight to Amsterdam, or take a flight to Paris, followed by a bus to Amsterdam”, for example, is a partial program. This program specifies a partial plan in which the first step (i.e. taking the GH123 bus) is fully specified, while the second and possibly third steps are only partially specified. This partial plan, when used as a search template, has some desirable effects on the search process. On the one hand, it helps the planner cut down on its search tremendously. For example, any plan that does not start with a GH123 bus ride can be immediately eliminated. Also, any plan that does not involve an Amsterdam-bound or a Paris-bound flight as the second step can also be immediately eliminated. On the other hand, it is not overly restrictive, and still leave the planner with a lot of flexibility to make its decisions. In proposing the second step, for example, the planner is free to choose any Amsterdam-bound or Paris-bound flight from Toronto that it deems appropriate. As can be seen, the amount of search a planner needs to perform in coming up with a plan can be controlled by making the search template more or less specific. This is the main idea behind partial programming. By allowing the system developer to express his or her procedural insights (which is valid but often incomplete), as partial programs, and by using these partial programs as search templates for the planner, one can
help the planner trims its search process to a much smaller size, and therefore comes up a plan much faster.

In our framework, partial progaming is supported via a set of language constructs provided by the Planning Ontology. Figure 3.7 shows these constructs and their relationships.

As shown in the figure, the main concept for expressing heuristic Advice is the Advice concept. To provide the planning engine with a heuristic advice, the application developer would instantiate an Advice object, and populate it with appropriate property values:

- The TriggerCondition property is populated with an object of type
LogicalExpression that represents the logical condition in which the advice is applicable.

- The ActionTemplate property is populated with an object of type PartialProgram representing the partial program that should be used as a search template whenever the TriggerCondition holds true. As shown in the figure, this PartialProgram object can be one of several types:

  - **Sequence** (i.e. A sequential composition of two subprograms): A partial program of type Sequence represents an advice to the planner that it should concatenate SubProgram1, which is a partial program itself, with SubProgram2, which is also another partial program, and use the resulting program as the search template.

  - **Conditional** (i.e. An If-Then-Else branching between two subprograms): A partial program of type Conditional represents an advice to the planner that it should select SubProgram1 as the search template if the Condition property holds. Otherwise, it should select SubProgram2 as the search template.

  - **Loop** (i.e. Repeated execution of a subprogram): A partial program of type Loop represents an advice to the planner that it should keep using SubProgram as the search template for as long as the Condition property holds true.

  - **Iteration** (i.e. For-each loop): A partial program of type Iteration represents an advice to the planner that it should use SubProgram as the search template once for each arguments in the ArgumentList.
- **Choice** (i.e. An alternative between two subprograms): A partial program of type *Choice* represents an advice to the planner that it should try using the subprogram specified by *SubProgram*1 as the search template. If that is not possible, use the subprogram specified by *SubProgram*2 as the search template.

- **SpecificAction**: A partial program of type *SpecificAction* represents an advice to the planner that it should select the action specified in the *Action* property. Unlike all the other types above, a *SpecificAction* is a non-recursive template that will result in a specific action being selected by the planner.

- **Search**: A partial program of type *Search* represents a request to the planner asking it to search for a set of values that, if assigned to the free variables that appears in *Expression*, will make it true. *Search* partial programs are often used in preceding of *SpecificAction* to help selecting the appropriate action parameters for the action selected by *SpecificAction*.

We elaborate on these constructs using an example. In Section 2.2 we discussed an example heuristic for the trip planning problem, i.e. “Whenever the traveler is in a city without an airport, he or she should take a bus to the destination directly. Or, if this is not possible, take a bus to a reachable airport”. This heuristic could help prevent the traveler from wandering between isolated (i.e. non-airport) cities, where the chance of reaching his or her destination is smaller. Figure 4.8 on page 97 shows how the *Advice* and *PartialProgram* concepts are used to represent this procedural insight as a heuristic advice to
the planner. As can be seen in the figure, the Advice object has two parts, a trigger expression that specifies the condition in which the advice is applicable (i.e. The user is in a non-airport city), and a partial program that could be used as a search template when this condition holds. This partial program is a Choice between two branches. The first is a Sequence of a Search, which asks the planner to search for a bus route (i.e. a value for BusNo variable) that goes from the current city to the destination, and a SpecificAction, which will actually select the TakeBus action if such bus route was found. If such a bus route doesn’t exist, Search will fail, and the second branch of the Choice is used as the new search template. This second branch is also a Sequence of a Search, which asks the planner to search for an airport that is reachable from the current city by bus (i.e. the values for AirportCity and BusNo variables), and a SpecificAction, which actually selects the bus ride.\footnote{For simplicity, we assume that all cities are reachable by bus from at least one airport city, and so this second subprogram should always succeed. Should it become necessary, this assumption can be eliminated by adding more logics into the search template.}

In Section 4.1.4, we will revisit these heuristic constructs with a more complex example.

3.6 The Integration Framework: Translating and Executing ontological planning problem descriptions

In the previous section, we discussed the various language constructs that are available in the Planning Ontology for describing planning problems. In this

\footnote{For simplicity, we assume that all cities are reachable by bus from at least one airport city, and so this second subprogram should always succeed. Should it become necessary, this assumption can be eliminated by adding more logics into the search template.}
section, we describe how planning problem descriptions expressed using those constructs are translated and executed by the framework. We start out by describing the design of the translation engine. We then go on to describe how each of the language constructs discussed in the previous section are mapped into Horn clauses to make an executable rule-based planning program. Finally, we describe how the translated program can be executed using a Prolog-based planner to produce valid plans.

3.6.1 The Translation Engine

Figure 3.8: The software design of the translation engine.

Figure 3.8 shows the design of the translation engine. In the figure, the OWLAPI layer refers to the OWLAPI Library [54], a generic and open-source Java library for accessing, querying and manipulating OWL ontologies. The next layer, OWLAPI Facade, is an abstraction we built on top of the OWLAPI to make it simpler and more convenient to interact with the ontologies, and to “insulate” our code from any future changes to the API. Because OWLAPI is
a generic API designed to be used by the widest range of applications possible, the programming interface that it exposes is detailed and can be overly complex at times. Also, calling this API directly requires API-specific codes to be inserted in many places inside the caller, making it rigid and less maintainable against any future changes that might occur in the API. Calling the API through a facade allows us to work with a new programming interface that is simpler and much more convenient for our specific purpose of translating planning problem descriptions. It also prevents API-specific codes from permeating into our logics. The top layer, Translation Logic, is where the actual logic for translating ontological descriptions into Horn clauses belongs. In the remaining of this section, we will focus our discussion on the logic contained in this layer.

### 3.6.2 Translating planning problem descriptions

In this section, we go into the details of translating planning problem descriptions made by the application developer into executable Prolog programs (i.e. KBs).

**Translating Logical Expressions**

We recall that the logical expressions are ontological expressions that are built from the language constructs described in Figure 3.5. The planning engine translates all logical expressions using a recursive procedure. This procedure takes as its input a `LogicalExpression` object, and outputs a Prolog clause according to the type of the input object:
• ComparisonConstraints: If the input expression is one of the ComparisonConstraints objects, the procedure will simply output a Prolog string of the form

$Operand1 $Comparator $Operand2

e.g.

City = guelph
CreditCount >= 20

where $Comparator can be any Prolog comparator symbol such as =, <, >, etc., while $Operand1 and $Operand2 are the variables or constants specified by the expression.

• RelationalConstraint: If the input expression is a RelationalConstraint object, the procedure will output a Prolog string of the form

$PredicateName($Arguments)

e.g.

destCity(paris)
airportCity(City)
busRoute(BusNo, guelph, toronto)

where $PredicateName is the name of the relation specified in the expression, and $Arguments are the constants or variables specified in the input object.

• Conjunction, Disjunction, Negation: If the input expression is a
Conjunction, Disjunction or Negation, the procedure will output a Prolog string of the form

$\text{SubExpression}_1 , \text{SubExpression}_2 , ...$

$\text{SubExpression}_1 ; \text{SubExpression}_2 ; ...$

not ( $\text{SubExpression}$ )

where the comma, semicolon, and not are Prolog symbols for AND, OR and NOT logical operators, and $\text{SubExpression}_N$ are the recursive Prolog strings that are obtained by calling the procedure recursively on the subexpressions that are specified in the input.

Translating Planning Problem Description

We recall that the description of a planning problem consist of ontological expressions that representing the basic ingredients of the that problem. These ontological expressions, which are built using the language constructs described in Figure 3.6, are translated by the planning engine as follows:

• Actions: All expression of type Action are translated into a Prolog assertions of the form:

  action($\text{ActionName}(\text{Arguments}))$.

  e.g.

  action(getCoffee).
  action(drinkCoffee).
  action(takeBus(BusNumber)).
$ActionName$ is the name of the action, and $Arguments$ is a comma separated list of variables representing the arguments of the $Action$ object. The three example assertions above state that $getCoffee$, $drinkCoffee$ and $takeBus(BusNo)$ are the planning actions from which the planner can select to construct a plan.

- $PreCondition$: All expressions of type $PreCondition$ are translated into a Prolog statements of the form:

  \[\text{poss}(\text{$Action$, $S$}) :\text{-} \text{$LogicalExpression$}.\]

  e.g.:

  \[\text{poss}(\text{takeFlight(FlightNo), S}):\text{-}\]
  \[\text{flight(FlightNo, SrcCity, DstCity),}\]
  \[\text{inCity(City, S),}\]
  \[\text{City = SrcCity.}\]

  where $Action$ is the name of the action, together with its arguments, $S$ is the Prolog list representing the current situation, and $LogicalExpression$ is the Prolog string generated from the logical expression in the $PreCond$'s LogicalExpression property.

  The example precondition above states that it is possible to take a given flight if its source city is the same as the traveller’s current location. Readers who are familiar with SitCalc will recognized that this statement is a $PreCondition$ Axiom in SitCalc.

- $StateFeatures$: All expressions of type $StateFeature$ are translated into a pair of Prolog statements of the form:
$\text{FeatureName}($NonSituationArguments, [A|S]):-
   $\text{DefiniteValueClause};$
   $\text{InvariantClause},$
   $\text{FeatureName}($NonSituationArguments, S).

and

$\text{FeatureName}($NonSituationInitialValues, []).

e.g.

hasCoffee([A|S]):-
   A = getCoffee;
   not (A = drinkCoffee),
   hasCoffee(S).

hasCoffee([]).

where $\text{FeatureName}$ is the name of the feature, $\text{DefiniteValueClause}$
is the Prolog string generated from the logical expression in the feature’s
$\text{DefiniteValueCond}$ property, $\text{InvariantClause}$ is the Prolog string
generated from the logical expression in the feature’s $\text{InvariantCond}$
property, and $\text{NonSituationArguments}$ is a comma separated list of
Prolog variables representing all the arguments of the state feature, ex-
cept its situation argument. The situation argument itself is translated
into [A|S], which is a list in Prolog, where A represents the last action,
and S represents the previous situation before A was taken. (We recall
from our background discussion in Section 2.3 that SitCalc $\text{Situations}$
are commonly represented in Prolog as reversed lists, where the first list
element represents the last action in the situation, the second list ele-
ment represents the action before that, and so on. Also, an empty list, [], represents the InitialSituation, which is the starting state in which no action has been taken yet). Lastly, NonSituationInitialValues is the list of initial values for all the arguments of the state feature. These values are extracted from the arguments’ InitialValue property.

In the example, the hasCoffee([A|S]) state feature can be read as “The traveller is considered to have a coffee in a given situation if the last action, A, is getCoffee, or if he or she had a coffee in the previous situation, S, and the last action is not drinkCoffee. Also, the traveler is assumed to have started with coffee in hand.”.

Reader who are familiar with SitCalc will recognize that this statement is a Successor State Axiom.

Translating Heuristic advice

We recall that heuristic advice is expressed as using the Advice concept, discussed in Figure 3.7. These expressions are translated into Prolog tertiary assertions of the form:

```prolog
advice(
    $AdviceName,  
    $TriggerCondition,  
    $PartialTemplate
).
```

where $AdviceName is the name of the advice, $TriggerCondition is the Prolog string generated from the logical expression in the Advice’s
TriggerCondition property, $PartialTemplate$ is the Prolog string that is translated from the Advice's ActionTemplate property as follows:

- **Sequence**: All expressions of type Sequence are translated into a Prolog string of the form

  $$\text{$SubProgram1 : $SubProgram2}$$

  where $SubProgramN$ are the Prolog strings generated by recursively translating the Sequence's sub-programs.

- **Conditional**: All expressions of type Conditional are translated into a Prolog string of the form

  $$\text{if($Condition, $SubProgram1, $SubProgram2)}$$

  where $SubProgramN$ are the Prolog strings generated by recursively translating the Conditional's sub-programs, and $Condition$ is the Prolog string generated by translating the logical expression in the Condition property.

- **Loop**: All expressions of type Loop are translated into a Prolog string of the form

  $$\text{while($Condition, $SubProgram)}$$

  where $Condition$ is the Prolog string generated by translating the logical expression in the Loop's Condition property, and $SubProgram$ is the Prolog string generated by recursively translating the Loop's sub-program.
• **Choice**: All expressions of type *Choice* are translated into a Prolog string of the form

\[ \$\text{SubProgram1} \# \$\text{SubProgram2} \]

where \( \$\text{SubProgramN} \) are the Prolog strings generated by recursively translating the *Sequence*’s sub-programs.

• **SpecificAction**: All expressions of type *SpecificAction* are translated into a Prolog string of the form

\[ \$\text{ActionText} \]

where \( \$\text{ActionText} \) is the Prolog strings representing the action name and its arguments.

• **Search**: All expressions of type *Search* are translated into a Prolog string of the form

\[ \text{search}(\$\text{LogicalExpression}) \]

where \( \$\text{LogicalExpression} \) is the Prolog strings generated by translating the logical expression in the *Search*’s *Expression* property.

Using the translation mappings, the example heuristic we discussed in Section 3.5.3 are translated into the following SitCalc statement:

\[
\text{advice}(
    \text{advice1},
    \text{inCity}(\text{City}, \text{S}) \& \neg \text{airport}(\text{City}),
    ()
    \text{search}(
\]


destCity(DestCity) & bus(BusNo, City, DestCity)
)
)

). takeBus(BusNo)
#
search(
bus(BusNo, City, Airport) & airportCity(Airport)
)

). takeBus(BusNo)

)

In the listing, the triggering condition is that the traveler is in a non-airport city, and the search template is a Choice between taking the bus to the destination, and taking the bus to a nearby airport city. The symbols &, |, and ~, which appear in the trigger condition, are operators understood by the SitCalc planner as logical symbols for AND, OR and NOT, respectively, while the symbols : and #, which appear in the action template, are operators understood by the SitCalc planner as sequential composition and alternative choice, respectively.

3.6.3 Executing User’s Requests

Once all the Actions, PreConditions, StateFeatures, and heuristic Advice have been translated, they are ready to be used for answering user requests.

When the user makes a planning request, he or she will describe his or her objective to the application. This is done by the user instantiating an object
of type *ObjectiveStatement*, shown in Figure 3.9, and populate its properties with appropriate values as follows:

- **Overrides** – This property can simply be assigned a Null value if the user just wants to start the planning process from default state feature values. If the user wants to start with non-default state feature values, this property can be populated with *RelationalConstraint* objects (explained in Figure 3.5) representing the overriding assertions. As an example, Figure 4.9 shows how the values of the starting and destination cities are overridden using an objective statement.

- **GoalExpression** – This property is populated with a *LogicalExpression* (explained in Figure 3.5) representing the logical description of the goal state. The example in Figure 4.9 also shows an example logical description of the goal state.

![ObjectiveStatement](image)

**Figure 3.9:** The structure of the *ObjectiveStatement* concept

Upon receiving the user’s planning request, the system services it by first translating the input *ObjectiveStatement* object into Prolog statements, just as it would with other *RelationalConstraint* and *LogicalExpression* objects, merge them into the main knowledge base along with the Actions, Pre-Conditions, State Features, and execute the merged KB on a SitCalc planner (e.g. the Golog interpreter) to obtain a plan and return it to the user.
3.7 Chapter Summary and Discussions

As a summary, we started this chapter by presenting the primary rationales for a translation approach to supporting planning in ontology-driven applications (Section 3.1). We argued that such an approach would offer several advantages over existing language-based and parallel modeling approaches, including enhanced reusability of knowledge, framework independence, reliance on mature and proven formalisms and tool sets, non-intrusiveness and user familiarity, etc. Next, we described the fundamental challenges (i.e. representability, translatability) that such an approach would have to overcome in order to be successful, and made two important observations that would allow us to overcome those challenges (Section 3.2). We also provided a thesis statement that concisely describe our research objectives, and outlined the research methodology that we will follow (Section 3.3). We then moved on to describe our proposed architecture, its main components and the overall process that an ontology-driven application developer could follow to build his or her application using this architecture (Section 3.4). Finally, we provided a detailed description of the language constructs that are offered by the Planning Ontology (Section 3.5), as well as the design and operation of the translation engine (Section 3.6).

In this section, we conclude the chapter with some additional discussions regarding the translatability and correctness of the ontological models that are possible through the Planning Ontology, as well as the strengths, limitations and formal properties of our proposed framework.
3.7.1 Translatability of Ontological Models

In Sections 3.5 and 3.6 above, we discussed all the language constructs provided by the Planning Ontology in details as well as how they are translated into Horn (i.e. Prolog) statements. In this section we discuss the translatability of the ontological models that are possible through these language constructs. We will show that all ontological expressions created by the application developer using the language constructs from the Planning Ontology are translatable by our framework using the mappings we have discussed earlier in this section. We do this by going over each object types (i.e. concepts) from the Planning Ontology, and show that they are always translatable using the provided mappings.

Translatability of Logical Expressions

To show that all objects of type LogicalExpression are always translatable by our framework, we make the following observations:

- First, we consider objects of type ComparisonConstraint, i.e. Equal, GreaterThan, LessThan, etc. These objects contains two inner objects representing their operands, where these operand objects can be one of the three types:

  - A literal constant. e.g. 25 or “guelph”, etc.
  - An object of type Variable.
  - An object of type Action.
When the operand is a literal constant or a Variable, it is trivially translatable. When the operand is an Action, its translatability depends on the translatability of its PreCondition object. But because the defined mapping for ComparisonConstraint ignores the action's PreCondition, this type of operand is translatable regardless. As such, all three operand types are translatable. Therefore, we can conclude that all ComparisonConstraint objects are translatable, because their operands are always translatable.

- Second, we consider objects of type RelationalConstraint. These objects are always translatable because all they contain are a relation name, which is a string literal, and one or more arguments, which are either variables or literal constants, all of which are translatable.

- Third, we note that objects of types Conjunction, Disjunction and Negation are always translatable using the provided mappings as long as the inner objects representing their subexpressions are translatable.

With the three results above, we can conclude, via structural induction, that all objects of type LogicalExpression are translatable – The first two results above serve as the base cases, whereas the third result serves as the inductive case.

Translatability of Planning Concepts

Because all LogicalExpressions are translatable, all objects of type Action, StateFeature, GroundFact, and ObjectiveStatement are always translatable using the provided mappings.
Translatability of Heuristic Advice

Using the same argument we used to prove the translatability of LogicalExpressions, we can also prove the translatability of all objects of type PartialProgram. The base cases are the SpecificAction and Search concepts, while the inductive cases are all the remaining PartialProgram concepts.

Also, since all PartialPrograms are translatable, all Advices are also translatable.

3.7.2 Correctness of Ontological Models

In the preceding section, we showed that all ontological expressions that the application developer can make through the concepts in the Planning Ontology are translatable. As such, by using our framework, the application developer is guaranteed to produce translatable and executable planning programs. This guarantee, however, does not cover the correctness of the programs. That is, the framework is not able to ensure that the application developer will always produce models/programs that will accomplished his or her desired outcomes. Using the provided language constructs, the user is guaranteed to make syntactically correct models, but its actually semantic can still be different from what he or she intended.

While this problem can not be eliminated easily, it can be alleviated via the introduction of sophisticated checks into the tooling facility. These checks can help detect common modeling mistakes and bring them to the developer’s attention, giving him or her a chance to see and correct any potential problems.
As we have explained in Section 3.5 and Section 3.6, the language constructs in our Planning Ontology are based closely on the concepts of the Situation Calculus (SitCalc) language – A planner-independent logical specification language for describing dynamically changing worlds (Please see Section 2.3 for some background discussions). Using SitCalc as a foundation for the Planning Ontology afforded us two important benefits. The first major benefit is that it allows our framework to immediately inherit the primary strengths of SitCalc. In particular:

- **Planner Independence** – This advantage allows the application developer to describe his or her planning problems in a framework and planner independent way, and thus ensuring a larger chance of reusability.

- **Effective Control of the Planning Process** – This advantage allows the ontology-driven application developer to be able provide procedural heuristic Advice to the planner (in the form of partial programs) to speed up the planning process, and thus increase the application’s effectiveness in handling the complexity associated with real-world planning problems.

The second benefit of using SitCalc as a foundation for the Planning Ontology is that the types of planning problems that can be described by our framework is already well-understood. In particular, the language supported by our Planning Ontology is equivalent to what referred to as the “Basic Action Theory” in [93], and as such can be used to describe planning problems that are:
• Deterministic: All actions have deterministic (i.e. non-stochastic) effects on the environment.

• Linear: Only one action are to be selected at the same time (i.e. no concurrency)

• Instantaneous: All actions can be assumed to have no duration.

• Static: The environment’s dynamics (i.e. how it responses to an action being performed) remain static through out the planning process.

Planning problems that needs to handle stochasticity, concurrency, temporal constraints, etc., will require additional vocabulary to be added to the Planning Ontology, and thus can not be described using our language at the present time. While this might sound a bit too restrictive at first, there are two reasons why this is not a problem. First, many useful real-world planning problems do not exhibit any of these characteristics (See, for example, the second case study in Chapter 4). Second, these restrictions can be removed by adding more vocabulary to the Planning Ontology. To our best knowledge, this extension should be a relatively straight forward exercise, as SitCalc itself has already had extensions that deal with these specific issues.

This section concludes our chapter. In the next chapter, we will describe how we applied the proposed framework to build two interesting ontology-driven intelligent systems.
Chapter 4

Case Studies and Evaluation

As we have discussed in the previous chapter, the primary objective of this thesis is to prove the feasibility of a translation-based approach to the integration of planning into ontology-driven applications. In particular, we identified, in Section 3.3 three specific questions that need to be answered:

- **Feasibility** – Is it actually possible to overcome the representability and translatability challenges (described in Sections 3.2.1 and 3.2.2 respectively) to build the proposed integration framework?

- **Effectiveness** – Is the resulting framework effective (i.e. powerful and flexible) enough to handle complex, real world planning problems?

- **Reusability** – Can the resulting framework be reused across applications and domains without requiring extensive modifications?

In Sections 3.2.1 and 3.7.1 we answered the theoretical component of the Feasibility question by showing that:
1. We can overcome the representatibility challenge and provide a way for the system designer to represent Horn clauses in Description Logics: This can be accomplished using a set of ontological constructs representing logical operators (Section 3.2.1).

2. We can overcome the translatability challenge and ensure that all planning problem descriptions produced by the system designer are translatable to rule-based planning programs: All ontological expressions produced by the application designer using our proposed framework are inductively guaranteed to be translatable to equivalent Horn clauses (Section 3.7.1).

In this chapter, we examine the experimental component of the Feasibility question by applying our framework to solve a simplified version of the Intelligent Trip Planner problem mentioned in previous chapters. This exercise requires us to construct and exercise the proposed framework from end to end, and serves as an experimental proof for the feasibility of our framework.

We also answer the Effectiveness and Reusability questions by applying our framework to the problem of building an Intelligent Student Advisor, which provides students with real-world course selection recommendations based on the student’s stated goal. As will be explained in details in Section 4.3 below, this problem is much more complex as compared to the trip planner problem, and its success will help demonstrate the framework’s effectiveness in coping with complexity in real-world planning problems. Also, because its domain is very different from that of the trip planning problem, it success also help demonstrate the reusability of our framework.
4.1 Case Study 1: An Intelligent Trip Planner

This case study serves as the feasibility study of our proposed approach. While it has been intentionally kept simple, for clarity and ease of explanation, this case study presents both the representability and translatability challenges described in Section 3.2 and exercises our framework from end to end to uncover any weakness that it might have.

4.1.1 Problem Formulation

In this case study, we apply the proposed framework to build the Intelligent Trip Planner described in Section 2.2. This system provides its users with suggestions on how to get from one place to another using a combination of flights, train and bus routes. As a planning problem, the trip planner can be formulated as follows:

List of Actions

There are 6 different actions.

- \texttt{takeFlight(FlightNumber)}.
- \texttt{takeTrain(TrainNumber)}.
- \texttt{takeBus(BusNumber)}.
- \texttt{getCoffee()}.
- \texttt{drinkCoffee()}.
• takeRest().

It can be noted that among the six actions above, takeFlight(), takeTrain() and takeBus() are parametrized whereas getCoffee(), drinkCoffee() and takeRest() are parameterless. The main difference is that when the planner enters the deliberation process to select the next action for the travel plan, each of the parameterless actions represents a single choice, whereas the parametrized actions each represents a set of choices (one for each available flight, bus and train number in the knowledge base).

List of State Features

For the purpose of our case study, we assume that the travel planning world has only 3 features, and hence its states can be completely characterized using the triple \(<HasCoffee, IsRested, InCity>\), where the three variables are as follows:

• HasCoffee – This is a binary variable representing whether or not the traveler is holding a coffee. Intuitively, the value of this variable will be equal to true after the action getCoffee() is performed, and equal to false after the action drinkCoffee() is performed. Initially, when neither of these actions has been performed, the variable is assumed to have the value of false.

• IsRested – This is another binary variable representing whether or not the traveler is rested (i.e. not tired or sleepy). Intuitively, the value of this variable will be equal to true after the action takeRest(), and false
after either \textit{takeFlight()} or \textit{takeTrain()}. Initially, when no action has occurred, the value is assumed to be \textit{true}.

- \textit{InCity} – This is a discreet variable representing the current location (i.e. city) of the traveler. Intuitively, the value of this variable is equal to the destination city of the flight, bus, or train after the traveler take one of these actions. When none of these actions has been taken yet, the value is equal to the traveler’s initial city.

From a logical (i.e. SitCal) perspective, these three feature variables can be represented as logical expressions on the situation term as follows:

- \textit{HasCoffee}(S)
- \textit{IsRested}(S)
- \textit{InCity}(City, S)

In these expressions, \( S \) is the situation variable (i.e. a variable representing a state that occurs as the result of a particular sequence of actions – please see section 2.3 for a more detailed description of the notion of situation). These expressions represent the state features in a logical way. In section 4.1.4 below, we will describe how the dynamics of these state features, can be described using the language constructs in the Planning Ontology.

**Action Preconditions**

We also assume the following pre-conditions exist for the six actions.
• \textit{takeFlight}(\textit{FlightNumber}), \textit{takeTrain}(\textit{TrainNumber}) and \textit{takeBus}(\textit{BusNumber}) are possible only if the traveler is currently located in the same city as the source city of the flight/train/bus.

• \textit{getCoffee}() is possible only if the traveler is not holding a coffee.

• \textit{drinkCoffee}() is possible only if the traveler is holding a coffee.

• \textit{takeRest}() is always possible.

**Initial and Goal State**

The default starting state is assumed to be one in which the traveler is located in Guelph, well-rested and does not have a coffee. As described in Section 3.6.3 these defaults can be overridden by the user on a per-request basis.

The goal state description will also be provided by the user when he or she makes the planning request to the system, on a per-request basis. We explain how this can be done in Section 4.1.7.

**Heuristic Advice**

As the designer of the trip planner, we can also supply the framework with some heuristic advice on how it can speed up the planning process. An example of such advice, mentioned in Section 3.5.3 is a rule that could help prevent the traveler from wandering between isolated (i.e. non-airport) cities, where the chance of reaching his or her destination is smaller: “Whenever the traveler is in a city without an airport, he or she should take a bus to the destination directly. Or, if this is not possible, take a bus to a reachable airport”.

84
In Section 4.1.4 below, we will show how the language constructs in the Planning Ontology can be used to encode this advice as a partial program.

4.1.2 Application of the proposed architecture

With the trip planner formulated as a planning problem, it can be built using our proposed framework. Figure 4.1 illustrates the main components that the application developer would need to construct in order to build the Trip Planner using our proposed framework.

Figure 4.1: Application of our proposed framework to the construction of the intelligent trip planner. The Planning Ontology (described in Section 3.5) and the Translation Engine (described in Section 3.6) are supplied as parts of our framework. The Trip Planning Objectives Ontology, Travel Ontology and KB, and the Trip Planning KB are to be supplied by the application developer. The Goal Statement is supplied by the end user on a per-request basis.

As shown in the diagram, in order to build the intelligent planner using
our framework, the application developer would need to provide three main artifacts:

- **The Travel Ontology and KB** – These two artifacts, shown in green in the figure, contain all the relevant knowledge about the various flights, train and bus routes, cities, airports, etc. The Travel Ontology and KB are described in details in Section 4.1.3 below.

- **The Trip Planning KB** – This KB, shown in blue in the figure, contains the ontologically description of the planning problem that underlies the trip planning process. In this ontology, the application developer would encode all the planning-related knowledge needed by the planning engine to compute valid travel plans for the user. In Section 4.1.4 below, we will describe in details how the developer could do this using the various language constructs from the Planning Ontology.

- **The Objective Ontology** – This ontology, shown in purple in the figure, provides the users (i.e. travelers) with the necessary language by which to express their travel planning objectives. We describe this ontology in Section 4.1.7 below.

### 4.1.3 Describing the Domain Knowledge: The Travel Ontology and KB

As the first step of building an application using our framework, relevant knowledge pertaining to the domain in which the application operates needs to be modeled. For our Trip Planning application, this domain knowledge is
modeled (i.e. described) in the Travel Ontology and KB. These two artifacts contain all the relevant knowledge of the travel domain. In particular, they contain the concept definition and assertions regarding cities, airports, flights, train routes, bus routes, hotels, etc. Tables 4.1 to 4.4 show some of the selected facts from the Travel KB that are relevant to our discussion in this section.

Table 4.1: Some selected flights from the Travel KB

<table>
<thead>
<tr>
<th>FlightNo</th>
<th>SrcCity</th>
<th>DstCity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac123</td>
<td>toronto</td>
<td>paris</td>
</tr>
<tr>
<td>ac223</td>
<td>paris</td>
<td>toronto</td>
</tr>
<tr>
<td>aa100</td>
<td>newyork</td>
<td>london</td>
</tr>
<tr>
<td>aa200</td>
<td>london</td>
<td>newyork</td>
</tr>
<tr>
<td>af100</td>
<td>paris</td>
<td>nantes</td>
</tr>
<tr>
<td>af200</td>
<td>nantes</td>
<td>paris</td>
</tr>
</tbody>
</table>

Table 4.2: Some selected train routes from the Travel KB

<table>
<thead>
<tr>
<th>TrainNo</th>
<th>SrcCity</th>
<th>DstCity</th>
</tr>
</thead>
<tbody>
<tr>
<td>rf10</td>
<td>nantes</td>
<td>paris</td>
</tr>
<tr>
<td>rf20</td>
<td>paris</td>
<td>nantes</td>
</tr>
</tbody>
</table>

Table 4.3: Some selected bus routes from the Travel KB

<table>
<thead>
<tr>
<th>BusNo</th>
<th>SrcCity</th>
<th>DstCity</th>
</tr>
</thead>
<tbody>
<tr>
<td>go100</td>
<td>guelph</td>
<td>toronto</td>
</tr>
<tr>
<td>go200</td>
<td>toronto</td>
<td>guelph</td>
</tr>
<tr>
<td>pt100</td>
<td>paris</td>
<td>nantes</td>
</tr>
<tr>
<td>pt200</td>
<td>nantes</td>
<td>paris</td>
</tr>
</tbody>
</table>

One thing worth noting about the Travel Ontology is that, while it was constructed for our application, the way it describes travel-related knowledge makes it totally independent from the application itself, and therefore can be directly reused by other applications. As we have discussed in Chapter 3,
Table 4.4: Some selected airports from the Travel KB

<table>
<thead>
<tr>
<th>City</th>
<th>HasAirport</th>
<th>IsHubAirport</th>
</tr>
</thead>
<tbody>
<tr>
<td>toronto</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>paris</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>newyork</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>london</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>nantes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>niagara falls</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>rochester</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>brantford</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>peterborough</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>belleville</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>orangville</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>guelph</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

This ability to separate domain knowledge from planning-related knowledge (i.e., separation of concerns), making it application-independent and purpose-independent, is one of the main benefits of our proposed architecture.

4.1.4 Describing the underlying planning problem: The Trip Planning KB

With the domain knowledge represented in Travel Ontology and KB, we can now move on to the second component of the application – The Trip Planning KB, shown in green in Figure 4.1. As mentioned earlier, this KB describes the planning problem that underlies the automated trip planning process. There are 5 main types of expression in this KB:

- **Logical Relations**

- **Actions**

88
In this section, we explain how each of the 5 description types above are expressed using the language constructs provided by the Planning Ontology.

**Defining logical relations**

The logical description of the travel planning process as a planning problem involves the use of several logical relations such as $City(x)$, $AirportCity(x)$, $Flight(\text{flightNo}, \text{srcCity}, \text{dstCity})$, etc. As such, in order to describe the underlying planning problem, we need to define these relations in our ontological model. In our framework, logical relations are defined using the *GroundFact* concept provided by the Planning Ontology. Intuitively, a ground fact can be thought of as an instruction to the underlying framework on how to construct a new logical relation using the data from an existing concept from the domain ontology.

As an example, Figure 4.2 shows a ground fact that serves as a definition/instruction on how to construct a new tertiary relation $Flight(\text{flightNo}, \text{srcCity}, \text{CreditWeight})$ using data from the *Flight* domain concept. In the figure, *Flight* is a ground fact that defines a tertiary relation, namely $Flight(\text{flightNo}, \text{srcCity}, \text{dstCity})$, using the data from the *Flight* concept in the Travel Ontology. That is, for each Flight object found in the Travel KB, a triple of the form $Flight(\text{flightNo}, \text{srcCity}, \text{dstCity})$ is produced, where $\text{flightNo}$ is equal to the object’s *FlightNumber* attribute.
value, \textit{SrcCity} is equal to the object’s \textit{DepartureAirport} attribute value, and \textit{DstCity} is equal to the object’s \textit{DestinationAirport} attribute value. The ground fact’s \textit{SourceRelationName} attribute specifies the domain concept upon which the new relations is defined. The DataAccessor objects specify how the triple values are to be extracted from the Flight objects. The first accessor, for example, specifies that to get the value for the ground fact’s first argument, the Flight object’s \textit{FlightNumber} attribute should be accessed. We will explain the DataAccessor object’s \textit{IsCollection} and \textit{InnerAccessor} attributes in Figure 4.14 in the second case study.
Describing the Actions

As we have explained in Section 3.5, actions can be described using the Action concept from the Planning Ontology. Figure 4.3 shows how the 6 actions are described in the Trip Planning KB.

![Diagram showing the actions](image)

Figure 4.3: Description of the actions from the Trip Planner. The PreCondition attributes for each action will be described later in this section, and are shown as blank in this figure.

As shown in the figure, TakeFlight/Train/Bus are parametrized actions that take one argument, and the other three actions are parameterless. The preconditions for these actions are described separately in their own section below, and left as blanks in the figure.

Describing the State Features

In this section, we describe how the three state features in our Trip Planning application can be described using the constructs from the Planning Ontology.
We recall from our discussion in Section 3.5.2 that a state feature represents a specific aspect of the environment’s state. As actions are performed against the environment, the state features change their values to reflect the new states that result from these actions. Not all actions will cause all state features to change their values, however. Each feature, in fact, only responds to a certain conditions, and remains the same in all other circumstances. As an example, the \textit{InCity} feature, which represents the traveler’s current location, only response to (i.e. is affected by) the actions \textit{TakeFlight}, \textit{TakeTrain} or \textit{TakeBus} being performed. All other actions (e.g. \textit{GetCoffee}, \textit{DrinkCoffee}, \textit{TakeRest}, or any other actions had they existed), do not effect the value of this feature.

In our framework, we describe this dynamic (i.e. how a state feature changes its value in response to actions being performed) via two logical expressions.

- \textit{InvariantCond} – This expression specifies the circumstances in which the feature value will remain invariant (not affected by the actions performed).

- \textit{DefiniteValueCond} – This expression specifies the circumstances in which the feature value will take on a definite value.

As the first example Figure 4.4 shows how the \textit{HasCoffee()} state feature is described in the Trip Planning KB. In the figure, \textit{HasCoffee()} take on a definite value, \textit{true}, if the last action taken is \textit{GetCoffee}, and will remain unchanged as long as the last action taken is not \textit{DrinkCoffee}.
Figure 4.4: Ontological description of the HasCoffee() state feature.

Similarly, Figure 4.5 shows how the IsRested() state feature is described in the Trip Planning KB. In the figure, IsRested() take on a definite value, true, if the last action taken is TakeRest, and will remain unchanged as long as the last action taken is not TakeFlight.

Figure 4.5: Ontological description of the IsRested() state feature.

Lastly, Figure 4.6 shows how the InCity() state feature is described in the Trip Planning KB. In the figure, IsRested() take on a definite value, City,
if one of the following three conditions holds: 1) The last action taken is \(\text{TakeFlight}(\text{FlightNo})\), where the flight \(\text{FlightNo}\) has \text{City} as its destination, or 2) The last action taken is \(\text{TakeTrain}(\text{TrainNo})\), where the train route \(\text{TrainNo}\) has \text{City} as its destination, or 3) The last action taken is \(\text{TakeBus}(\text{BusNo})\), where the bus \(\text{BusNo}\) has \text{City} as its destination. On the other hand, if the last action taken is neither \(\text{TakeFlight}\), \(\text{TakeTrain}\) or \(\text{TakeBus}\), the \(\text{InCity}\) will remain unchanged and keeps it value from the previous situation.

**Describing Action Preconditions**

With all the relations and state features described in the Trip Planning KB, we can now move on to describe the actions’ precondition. We recall from our problem formulation, Section 4.1.1, that the precondition for \(\text{TakeFlight/Bus/Train}\) is that the user is in the departure city, whereas the precondition for \(\text{DrinkCoffee}\) is that the user has a coffee. Figure 4.7 shows how these preconditions can be described in the Trip Planning KB.

The action \(\text{TakeRest}\) is assumed to be possible in all situations, and so its precondition expression can simply left as null/blank.

**Planning Heuristics**

Figure 4.8 shows how the example heuristic mentioned in our problem formulation section (i.e. “Whenever the traveler is in a city without an airport, he or she should take a bus to the destination directly. Or, if this is not possible, take a bus to a reachable airport”) can be described using the \(\text{Advice}\) concept provided by the Planning Ontology.
In the figure, the *TriggeringCondition* is an object of type *LogicalExpression* which states that the user must be in a non-airport city. The *ActionTemplate* is an object of type *PartialProgram*. As we have discussed in Section [3.5.3](#), this partial program is a *Choice* between two branches. The first is a *Sequence* of a *Search*, which asks the planner to search for a bus
route (i.e. a value for BusNo variable) that goes from the current city to the destination, and a SpecificAction, which will actually select the TakeBus action if such bus route was found. If such a bus route doesn’t exist, Search will fail, and the second branch of the Choice is used as the new search template. This second branch is also a Sequence of a Search, which asks the planner to search for an airport that is reachable from the current city by bus (i.e. the values for AirportCity and BusNo variables), and a SpecificAction, which actually selects the bus ride. 

4.1.5 Capturing User’s Objective: The Objective Ontology

With the domain knowledge described in the Travel Ontology and the planning problem that underlies trip planning described in the Trip Planning KB,

---

1For simplicity, we assume that all cities are reachable by bus from at least one airport city, and so this second subprogram should always succeed. Should it become necessary, this assumption can be eliminated by adding more logics into the search template.
Figure 4.8: *Description of the example heuristic advice from the Trip Planner problem: “Whenever the traveler is in a city without an airport, he or she should take a bus to the destination directly. Or, if this is not possible, take a bus to a reachable airport”*

the last step in building our Trip Planner is to design the Objectives Ontology, shown in purple in Figure 4.1. This ontology provides the users (i.e. travelers) with the necessary language by which to express their objectives, and contains only one concept, *ObjectiveStatement*. As we have discussed
in Section 3.6.3, the ObjectiveStatement’s Overrides property can be populated with RelationalConstraint objects to override the default state feature values, while the GoalExpression can be populated with an object of type LogicalExpression representing the logical description of the goal state.

Figure 4.9: An example objective statement.

Figure 4.9 show how an example objective, “Get from Guelph to Paris, well-rested and coffee in hand”, can be captured using the ObjectiveStatement concept. In this statement, the Overrides property is populated with a RelationalConstraints object, which states that, for this particular planning request, the planner should assume that the user is initially from Guelph. The goal state expression, on the other hand, specifies that the user wants to end up in a state in which InCity state feature has the value of “Paris”, and both
HasCoffee and IsRested state features should be true. As we will see in later, this statement, together with the actions’ preconditions and the logical description of the feature variables, will allow the planning framework to figure out a valid travel plan that satisfies the user’s stated objective.

4.1.6 Translating the Planning Problem Description

In Section [3.6] we discussed the design of the Translation Engine, and how it translates the various constructs in the Planning Ontology into corresponding Prolog statements. In this section, we shows how the main components of the trip planning problem are translated into Prolog rules. The complete generated Prolog program is included in Appendix A.

Ground Facts

Each ground fact defined in the Trip Planning KB is translated into a set of Prolog assertions:

flight(ac123, toronto, paris).
flight(ac223, paris, toronto).

... 
trainRoute(rf10, nantes, paris).
trainRoute(rf20, paris, nantes).

... 
busRoute(go100, guelph, toronto).
busRoute(go200, toronto, guelph).

...
airportCity(toronto).
airportCity(paris).
...

Actions

The 6 actions are translated into the following Prolog statements:

\[
\begin{align*}
\text{action}(\text{takeFlight}(\text{FlightNo})). \\
\text{action}(\text{takeTrain}(\text{TrainNo})). \\
\text{action}(\text{takeBus}(\text{BusNo})). \\
\text{action}(\text{takeBreak}). \\
\text{action}(\text{getCoffee}). \\
\text{action}(\text{drinkCoffee}).
\end{align*}
\]

These statements are simple Prolog fact assertions that convey to the planning engine that it can select from 6 different sets of actions when composing a travel plan for its user.

State Features

The three state features are translated into the following Prolog rules:

\[
\begin{align*}
\text{hasCoffee}([A|S]) :- \\
A = \text{getCoffee}; \\
\text{hasCoffee}(S), \\
\text{not} A = \text{drinkCoffee}.
\end{align*}
\]

100
isRested([A|S]):-
    A = takeRest;
    isRested(S),
    not A = takeFlight(_).

inCity(City, [A|S]):-
    A = takeFlight(FN), flight(FN, _, City);
    A = takeTrain(TN), train(TN, _, City);
    A = takeBus(BN), bus(BN, _, City);
    inCity(City, S),
    not (A = takeFlight(_); A = takeTrain(_); A = takeBus(_)).

In this listing, the commas (,;) represent logical conjunction and can be read as “AND”, the semicolons (;) represent logical disjunction, and can be read as “OR”, and the “:-” symbols represent logical implication and can be read as “IF”. Also, the square brackets ([ ]) represent lists in Prolog, and [A|S] represents the situation (i.e. list of actions) in which A was the last action in the sequence.

The rule for hasCoffee(), for example, can be read as follows: The user is holding a coffee in a given situation if either: a) He/She has just performed the action getCoffee, or b) He/She had a coffee in the previous situation, and didn’t drink it.
Action PreConditions

The action preconditions are translated into the following Prolog rules:

\[
\begin{align*}
\text{poss}(\text{getCoffee}, S). \\
\text{poss}(\text{drinkCoffee}, S) :&- \\
& \quad \text{hasCoffee}(S). \\
\text{poss}(\text{takeRest}, S). \\
\text{poss}(\text{takeBus}(\text{BN}), S) :&- \\
& \quad \text{busRoute}(\text{BN}, \text{Src}, _), \text{inCity}(\text{Src}, S). \\
\text{poss}(\text{takeTrain}(\text{TN}), S) :&- \\
& \quad \text{trainRoute}(\text{TN}, \text{Src}, _), \text{inCity}(\text{Src}, S). \\
\text{poss}(\text{takeFlight}(\text{FN}), S) :&- \\
& \quad \text{flight}(\text{FN}, \text{Src}, _), \text{inCity}(\text{Src}, S).
\end{align*}
\]

In this listing, \textit{hasCoffee()} and \textit{inCity()} are state features and \textit{flight()}, \textit{busRoute()}, \textit{trainRoute()} are ground facts. As we have discussed in previous section, these rules state that it is possible for the user to take a flight/-train/bus if he or she is currently located in the source city of the flight/bus/-train, drink a coffee if he or she has one, and get a coffee or take rest anytime.

Heuristic Advice

As we have discussed in Section 3.6.2, the example heuristic advice is translated into the following Prolog assertion:

\[
\text{advice(}
\begin{align*}
& \quad \text{advice1,}
\end{align*}
\]

102
\text{incity}(\text{City}, S) \land \neg \text{airport}(\text{City}), \\
( \\
\text{search(} \\
\quad \text{destCity}(\text{DestCity}) \land \text{bus}(\text{BusNo}, \text{City}, \text{DestCity}) \\
\quad ) : \\
\text{takeBus}(\text{BusNo}) \\
\quad # \\
\text{search(} \\
\quad \text{bus}(\text{BusNo}, \text{City}, \text{Airport}) \land \text{airportCity}(\text{Airport}) \\
\quad ) : \\
\text{takeBus}(\text{BusNo}) \\
) \\
).

In the listing, the triggering condition is that the traveller is in a non-airport city, and the search template is a \textit{Choice} between taking the bus to the destination, and taking the bus to a nearby airport city. The symbols $\&$, $\lor$ and $\neg$, which appear in the trigger condition, are operators understood by the SitCalc planner as logical symbols for AND, OR and NOT, respectively, while the symbols $:$ and $\#$, which appear in the action template, are operators understood by the SitCalc planner as sequential composition and alternative choice, respectively.
4.1.7 Executing User’s Requests

As we have discussed in Section 3.6.3, the system services user’s planning request by first translating the input *ObjectiveStatement* object, which might contains some *RelationalConstraints* as overrides and *LogicalExpression* describing the goal state, into a set of Prolog statements. The example objective shown in Figure 4.9 for instance, is translated into:

\[
\text{inCity(guelph, []).}
\]

\[
\text{goal(S):-}
\]

\[
\text{inCity(paris, S), isRested(S), hasCoffee(S).}
\]

In this listing, the empty list, [], represents the initial state (where no actions have been taken). The first assertion, therefore, states that the user is initially in Guelph. The last statement in the listing states that a situation S is considered a goal if the user has coffee, is rested and in Paris.

Once the input objective have been translated, they are merged into the main knowledge base along with Actions, PreConditions, State Features. The merged KB are then loaded and executed on a SitCalc planner (e.g. the Golog interpreter) to obtain a plan and return it to the user. Figure 4.10 shows a screen shot showing the execution of the generated program on the Golog planner, a standard Prolog-based SitCalc planner.

As can be seen in the figure, the objective of getting from Guelph to Paris, well-rested and coffee in hands can be accomplished, in the context of the simple model listed in tables 4.1 to 4.4 by a rather simple plan.
4.2 Case Study 2: An Intelligent Student Advisor

In this case study, we apply the framework to build a real-world planning system to demonstrate the framework’s effectiveness and reusability. Effectiveness is demonstrated through the complexity of the underlying planning problem, and reusability is demonstrated via the fact that the framework can be applied to this new application domain without modification.
4.2.1 Problem Formulation

Consider an intelligent student advising system, which we will call the Virtual Counselor (or VC for short), that could help university students select their courses based on their preferences and objectives. The VC’s job in life is to take an objective statement from a student (which could be something like “I want to complete my course work with at least 5 software engineering-related credits”, or “I want to take CIS*xxxx before my coop starts”, or a combination of objectives like these), and suggests an appropriate course-taking plan for the student, taking into account various constraints such as prerequisites, seasonal availability, workload, core course requirements, student’s semester levels, etc.

If we consider each course to be an action, then course selection can be seen as a planning problem, and as such our VC can be formulated as follows:

List of Actions

We will assume that there are two different types of actions:

- $takeCourse(CourseCode)$.
- $newSemester(SemesterLevel)$.

As its name suggests, $takeCourse(CourseCode)$ represents the act of taking a particular course. From a planning perspective, this action represents the set of choices (one choice per available course codes) from which a study (i.e., course taking) plan can be assembled.

$newSemester(SemesterLevel)$, on the other hand, is a hypothetical action that represents no physical action. Rather, it represents a choice by which the
reasoner can signify how it wants to divide the actual courses into semesters. For example, to suggest to the student that he or she should take CIS*1500 in the first semester, and CIS*1910 and CIS*2500 in the second semester, the VC would produce the following plan:

01: newSemester(1)
02: takeCourse(CIS*1500)
03: newSemester(2)
04: takeCourse(CIS*1910)
05: takeCourse(CIS*2500)

In the above plan, the newSemester(SemesterLevel) actions can be though of as “markers” that separate the actual courses into semesters. We will discusse the dynamics of the newSemester() action in more details in later parts of the case study. For the moment, it suffices to think of this action as a convenient way to structure the underlying planning problem.

List of State Features

Recall that the state features represent the different aspects of the environment, and together they characterize all the possible states of the reasoning problem. As compared to the Trip Planner in the previous case study, whose states were characterized by only three feature variables, the VC’s states need many more variables to be characterized, and each of these variables is also much more complex. We will discuss these variables below.

- AtSemesterLevel – This is a numerical (i.e., integer) variable representing the semester level (i.e., 1, 2, 3, ....) in which the student is in.
Intuitively, the value of this variable will increase after every occurrence of the `newSemester()` action. Initially, it is assumed to have the value of 1 (i.e., the students are assumed to start from the first semester), unless overridden by the user.

- **InSemester** – This is a categorical variable that can assume three different values – Fall, Winter, Summer. This variable represents the type of semester the student is in, and changes its value after every `newSemester` action. Initially, it is assumed to have the value `fall`, unless overridden by the user.

- **AtBeginningOfSemester** – This is a binary (true/false) variable that represents whether or not we are at the beginning of a semester. Intuitively, this variable assumes the value of `true` right after a `newSemester` action, and changes to `false` after a course has been selected afterward. Initially, the variable is assumed to have the value of `true`, unless overridden by the user.

- **Workload** – This is a numerical (i.e., real) variable that represents the student’s workload in the current semester. Intuitively, this variable has the value of 0 at the beginning of each semester, and increases with every course selected (by the selected course’s credit weight).

- **CreditCount** – This is a real-valued variable that represents the total credit the student have completed (since the beginning of the program). This variable is assumed to have the value of 0 at the beginning of the program, (unless overridden by the user), and increases in value every
time a course has been taken.

- **HasSelected[CourseCode]** – This is a composite variable that represents whether a course has been selected in the current semester. This variable can be thought of as an array of boolean variables, one for each available course code. Each of these boolean variable is assumed to have the value of *false* at the beginning of each semester, and changes its value to *true* if the corresponding course has been selected.

- **HasCompletedBy[CourseCode][SemesterLevel]** – This is a composite variable that represents whether or not a course has been completed prior to a given semester. Intuitively, this composite variable can also be thought of as a two-dimensional array of boolean variables, one for each available course code and semester level. These boolean variables has the value of *true* if the corresponding course has been completed (i.e., successfully taken) before the given semester, and *false* otherwise. Initially, all of these variables are assumed to have the value of *false*, unless overridden by the user.

- **HasCompleted[CourseCode]** – This is a derivative of the **HasCompletedBy** variable, and represents whether or not a course has been completed prior to the current semester. Intuitively, this composite variable can also be thought of as an array of boolean variables, one for each available course code. These boolean variables has the value of *true* if the corresponding course has been completed (i.e., successfully taken) before the current semester, and *false* otherwise. Initially, all of these variables are assumed to have the value of *false*, unless overridden
From a logical (i.e., SitCal) perspective, these feature variables can be represented as logical expressions on the situation term as follows:

- \( \text{AtSemesterLevel}(\text{Level}, S) \)
- \( \text{InSemester}(\text{Semester}, S) \)
- \( \text{AtBeginningOfSemester}(S) \)
- \( \text{Workload}(\text{Load}, S) \)
- \( \text{CreditCount}(\text{Count}, S) \)
- \( \text{HasSelected}(\text{Course}, S) \)
- \( \text{HasCompletedBy}(\text{Course}, \text{SemesterLevel}, S) \)
- \( \text{HasCompleted}(\text{Course}, S) \)

In these expressions, \( S \) is the situation variable (i.e., a variable representing a state that occurs as the result of a particular sequence of actions – please see section 2.3 for a more detailed description of the notion of situation). These expressions are called state expressions, and represent the state features in a logical way. In section 4.2.4 below, we will describe how these state expressions, together with their dynamics, can be described using the language constructs in the Planning Ontology.
Action Preconditions

The pre-conditions for our two actions are as follows:

• The action \textit{TakeCourse(CourseCode)} is possible if all of the following conditions are met:

  – The course is being offered in the current semester
  – The student has not been taken it in the past
  – The student has all the pre-requisites for the course
  – The workload does not exceed the maximum workload threshold (a configurable value).

• The action \textit{NewSemester(SemesterLevel)} is possible only if all of the following conditions are met:

  – The student has not exceed the maximum number of semesters allowed by the program (e.g., 12 semesters, a configurable value)
  – All the required (core) courses for the current semester have been selected.
  – The minimum workload requirement of the current semester has been satisfied (also a configurable value).

Initial and Goal State

The default starting state is assumed to be one in which the student is at semester level 0, in the fall, and has no prior credits. These defaults can be overridden by the students on a per-request basis.
The default goal state is one in which the student has completed at least 20 credits, at the same time satisfying the core courses requirements, course prerequisite, minimum and maximum workload, etc. On top of this basic requirement, the student can also specify additional objectives (e.g., the minimum number of software engineering courses, etc.) on a per-request basis.

**Heuristic Advices**

As the designer of the VC, we can also supply the framework with some heuristic advices on how it can speed up the planning process. An example of such advice in that if the VC found itself at the beginning of a semester, and there are some required courses for that semester, then it should immediately select all those courses before considering any other courses. By following this heuristics, the VC can save a lot of time considering the many course sequences that might end up not satisfying the core courses requirement. In Section 4.2.4 below, we will show how this advice can be easily provided to the framework using the language constructs in the Planning Ontology.

4.2.2 Application of the proposed architecture

With our Virtual Counselor formulated as a planning problem, it can be built using the proposed framework. Figure 4.11 illustrates the main components that the application developer would need to construct in order to build the VC application using our proposed framework.

As shown in the diagram, in order to build the Virtual Counselor using our framework, the application developer would need to provide three main
Figure 4.11: Application of our proposed framework to the construction of the Virtual Student Counsellor. The Planning Ontology (described in Section 3.5) and the Translation Engine (described in Section 3.6) are parts of the framework. The Course Objectives Ontology (purple), Course Ontology and KB (green), and the Course Selection KB (blue) are to be supplied by the application developer. The Goal Statement (orange) is supplied by the student on a per-request basis.

artifacts:

- **The Course Ontology and KB** – These two artifacts, shown in green in the figure, contain all the relevant knowledge about the various degrees, programs and courses offered by the university. They can be thought of as the ontological version of the university calendar, and is described in details in Section 4.2.3 below.

- **The Course Selection KB** – This KB, shown in blue in the figure, contains the ontologically description of the planning problem that underlies
the course selection process. In this ontology, the application developer would encode all the planning-related knowledge needed by the planning engine to compute valid course taking plans for the student. In Section 4.2.4 below, we will describe in details how the developer could do this using the various language constructs from the Planning Ontology.

- The Objective Ontology – This ontology, shown in purple in the figure, provides the users (i.e., students) with the necessary language by which to express their course-taking objectives. We describe this ontology in Section 4.2.5 below.

4.2.3 Describing the Domain Knowledge: The Course Ontology and KB

As the first step in building an application using our framework, relevant knowledge pertaining to the domain in which the application operates needs to be captured. For our Virtual Counselor application, this domain knowledge is captured (i.e., describe) in the Course Ontology and the Course KB. Figure 4.12 below shows the main concepts in this ontology.

As can be seen, the Course Ontology contains familiar concepts from the university calendar. In particular, Degree represents the different kinds of degrees (e.g., BComp) a student can earn, Program represents the different types of study programs (e.g., Computer Science, Software Engineering, etc.). Each program is thought to consist of several Semester Schedules, which specify what courses needed to be taken in each semester. Also, a course can have one or more Prerequisites, each of which can be either a single course, or a
choice between alternative courses.

For easy referencing, Table 4.5 shows CIS courses that are offered by the School of Computer Science, together with their credit weights and prerequisites.

Table 4.5: CIS courses contained in the Course KB

<table>
<thead>
<tr>
<th>Course</th>
<th>Credit</th>
<th>Offered In</th>
<th>PreRequisites</th>
</tr>
</thead>
<tbody>
<tr>
<td>cis1000</td>
<td>0.50</td>
<td>S, F, W</td>
<td></td>
</tr>
<tr>
<td>cis1200</td>
<td>0.50</td>
<td>F, W</td>
<td></td>
</tr>
<tr>
<td>cis1250</td>
<td>0.50</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>cis1500</td>
<td>0.50</td>
<td>F, W</td>
<td></td>
</tr>
</tbody>
</table>
Table 4.5: CIS courses contained in the Course KB

<table>
<thead>
<tr>
<th>Course</th>
<th>Credit</th>
<th>Offered In</th>
<th>PreRequisites</th>
</tr>
</thead>
<tbody>
<tr>
<td>cis1910</td>
<td>0.50</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>cis2030</td>
<td>0.50</td>
<td>F</td>
<td>cis1910, cis2500</td>
</tr>
<tr>
<td>cis2050</td>
<td>0.50</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>cis2170</td>
<td>0.75</td>
<td>W</td>
<td>(cis1200 or cis1500)</td>
</tr>
<tr>
<td>cis2250</td>
<td>0.50</td>
<td>W</td>
<td>cis1250, cis1500</td>
</tr>
<tr>
<td>cis2430</td>
<td>0.50</td>
<td>F</td>
<td>cis2500</td>
</tr>
<tr>
<td>cis2460</td>
<td>0.50</td>
<td>F</td>
<td>cis2500</td>
</tr>
<tr>
<td>cis2500</td>
<td>0.50</td>
<td>W</td>
<td>cis1500</td>
</tr>
<tr>
<td>cis2520</td>
<td>0.50</td>
<td>F</td>
<td>cis2500, cis1910</td>
</tr>
<tr>
<td>cis2750</td>
<td>0.75</td>
<td>W</td>
<td>cis2430, cis2520</td>
</tr>
<tr>
<td>cis2910</td>
<td>0.50</td>
<td>F</td>
<td>cis1500, cis1910</td>
</tr>
<tr>
<td>cis3000</td>
<td>0.50</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>cis3090</td>
<td>0.50</td>
<td>F</td>
<td>cis2030, cis3110</td>
</tr>
<tr>
<td>cis3110</td>
<td>0.50</td>
<td>W</td>
<td>cis2500</td>
</tr>
<tr>
<td>cis3120</td>
<td>0.50</td>
<td>W</td>
<td>cis2030</td>
</tr>
<tr>
<td>cis3150</td>
<td>0.50</td>
<td>F</td>
<td>cis2750, cis3490</td>
</tr>
<tr>
<td>cis3190</td>
<td>0.50</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>cis3210</td>
<td>0.50</td>
<td>F</td>
<td>cis3110</td>
</tr>
<tr>
<td>cis3250</td>
<td>0.50</td>
<td>F</td>
<td>cis2250, cis2500</td>
</tr>
<tr>
<td>cis3260</td>
<td>0.50</td>
<td>F</td>
<td>cis2430, cis2750, cis3250</td>
</tr>
<tr>
<td>cis3490</td>
<td>0.50</td>
<td>W</td>
<td>cis2910, cis2520</td>
</tr>
<tr>
<td>cis3530</td>
<td>0.50</td>
<td>F</td>
<td>cis2520, cis2750</td>
</tr>
</tbody>
</table>
Table 4.5: CIS courses contained in the Course KB

<table>
<thead>
<tr>
<th>Course</th>
<th>Credit</th>
<th>Offered In</th>
<th>Prerequisites</th>
</tr>
</thead>
<tbody>
<tr>
<td>cis3700</td>
<td>0.50</td>
<td>W</td>
<td>(cis3750 or cis3760), cis2460</td>
</tr>
<tr>
<td>cis3750</td>
<td>0.75</td>
<td>F</td>
<td>cis2750</td>
</tr>
<tr>
<td>cis3760</td>
<td>0.75</td>
<td>W</td>
<td>cis2750</td>
</tr>
<tr>
<td>cis4000</td>
<td>0.50</td>
<td>F, W</td>
<td></td>
</tr>
<tr>
<td>cis4050</td>
<td>0.50</td>
<td>F</td>
<td>cis2030, cis3110, cis3120</td>
</tr>
<tr>
<td>cis4110</td>
<td>0.50</td>
<td>W</td>
<td>cis3110</td>
</tr>
<tr>
<td>cis4150</td>
<td>0.50</td>
<td>F</td>
<td>(cis3750 or cis3760)</td>
</tr>
<tr>
<td>cis4210</td>
<td>0.50</td>
<td>W</td>
<td>(cis3750 or cis3760)</td>
</tr>
<tr>
<td>cis4250</td>
<td>0.50</td>
<td>F</td>
<td>cis2750, cis3260, cis3750</td>
</tr>
<tr>
<td>cis4300</td>
<td>0.50</td>
<td>F</td>
<td>cis3110, (cis3750 or cis3760)</td>
</tr>
<tr>
<td>cis4410</td>
<td>0.50</td>
<td>W</td>
<td>cis3210, (cis3750 or cis3760)</td>
</tr>
<tr>
<td>cis4430</td>
<td>0.50</td>
<td>W</td>
<td>cis3110, cis3530, (cis3750 or cis3760)</td>
</tr>
<tr>
<td>cis4450</td>
<td>0.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cis4500</td>
<td>0.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cis4650</td>
<td>0.50</td>
<td>W</td>
<td>cis2030, cis3110, cis3150</td>
</tr>
<tr>
<td>cis4720</td>
<td>0.50</td>
<td>W</td>
<td>cis2750, cis3110, cis2460</td>
</tr>
<tr>
<td>cis4780</td>
<td>0.50</td>
<td>F</td>
<td>(cis3750 or cis3760), cis3490, cis2460</td>
</tr>
<tr>
<td>cis4800</td>
<td>0.50</td>
<td>W</td>
<td>cis3110, (cis3750 or cis3760)</td>
</tr>
<tr>
<td>cis4820</td>
<td>0.50</td>
<td>W</td>
<td>cis3110, cis3750</td>
</tr>
<tr>
<td>cis4900</td>
<td>0.50</td>
<td>S, F, W</td>
<td></td>
</tr>
<tr>
<td>cis4910</td>
<td>0.50</td>
<td>S, F, W</td>
<td>cis4900</td>
</tr>
</tbody>
</table>
One thing worth noting about the Course Ontology is that, while it was constructed for our application, the way it describes course-related knowledge makes it totally independent from the application itself, and therefore can be directly reused by other applications. As we have discussed in Chapter 3, this ability to separate domain knowledge from planning-related knowledge (i.e., separation of concerns), making it application-independent and purpose-independent, is one of the main benefits of our proposed architecture.

4.2.4 Describing the Underlying Planning Problem: The Course Selection KB

With the domain knowledge represented in the Course Ontology, we can now move on the second component of the application – The Course Selection KB (shown in green in Figure 4.11). As mentioned earlier, this KB describes the planning problem that underlies the automated course selection process. There are 5 main types of expression in this KB:

- **Logical Relations**

- **Actions**

- **State Features**

- **Action Pre-Conditions**

- **Planning Heuristics**

In this section, we explain how each of the 5 description types above are expressed using the language constructs provided by the Planning Ontology.
Defining logical relations

The logical description of the course selection process as a planning problem involves the use of several logical relations such as \( \text{Course}(\text{CourseCode}, \text{CourseName}, \text{CreditWeight}) \) and \( \text{PreRequisite}(\text{CourseCode}, \text{Prerequisites}) \), etc. As such, in order to describe the underlying planning problem, we need to define these relations in our ontological model. In our framework, logical relations are defined using the \textit{GroundFact} concept provided by the Planning Ontology. Intuitively, a ground fact can be thought of as an instruction to the underlying framework on how to construct a new logical relation using the data from an existing concept from the domain ontology.

As an example, Figure 4.13 shows a ground fact that serves as a definition/instruction on how to construct a new tertiary relation \( \text{Course}(\text{CourseCode}, \text{CourseName}, \text{CreditWeight}) \) using data from the \textit{Course} domain concept.

In the figure, \textit{Course} is a ground fact that defines a tertiary relation, namely \( \text{Course}(\text{CourseCode}, \text{CourseName}, \text{CreditWeight}) \), using the data from the \textit{Course} concept in the Course Ontology. That is, for each course object found in the Course KB, a triple of the form \( \text{Course}(\text{CourseCode}, \text{CourseName}, \text{CreditWeight}) \) is produced, where \text{CourseCode} is equal to the object’s \text{CourseCode} attribute value, \text{CourseName} is equal to the object’s \text{CourseName} attribute value, and \text{CreditWeight} is equal to the object’s \text{CreditWeight} attribute value. The ground fact’s \textit{SourceRelationName} attribute specifies the domain concept upon which the new relations is defined. The \textit{DataAccessor} objects specify
Figure 4.13: The Course ground fact, which serves as an instruction on how to construct a new tertiary relation (i.e., Course(CourseCode, CourseName, CreditWeight)) using data from an existing domain concept of the same name (i.e., Course).

how the triple values are to be extracted from the Course objects. The first accessor, for example, specifies that to get the value for the ground fact’s first argument, the Course object’s CourseCode attribute should be accessed. The IsCollection and InnerAccessor attributes are explained in the next example.

As a more complex example, Figure 4.14 shows another ground fact that serves as a definition/instruction on how the binary relation PreRequisite(CourseCode, Prerequisite) can be constructed using data from the Course domain concept.

In the figure, PreReq is a binary relation that is defined on the Course class from the Course KB. As in the previous example, the first argument value
Figure 4.14: The PreRequisite ground fact, which serves as an instruction on how to construct a new relation, i.e. PreRequisite(CourseCode, PreRequisites) using data from an existing domain relation (i.e., Course).

can be extracted from the Course objects by accessing their CourseCode attribute values. The second argument, PreReqs requires a bit more work. First, its data accessor’s IsCollection attribute is set to true. This value tells the framework that the Course object’s PreRequisites attribute might contain multiple values (i.e., a course can have multiple pre-requisites), and hence it should try to access it as such. Second, this data accessor object also contains a nested inner data accessor. This inner accessor is needed because, unlike the CourseCode attribute, which contains simple data values, the PreRequisites attribute of the Course objects contains object (i.e., structured) values of type PreRequisite (Please refer back to Figure 4.12). To navigate this class struc-
ture and access the desired inner attribute value, an inner accessor is needed. Note that the inner accessor’s `IsCollection` value is also set to true. This is because each pre-requisite can be an alternative between multiple courses (Please also refer back to Figure 4.12).

Describing the Actions

As we have mentioned in the problem formulation, there are two types of actions in our system. We describe these actions using the Action concept from the Planning Ontology. Figure 4.15 shows how the 2 actions are described in the Course Selection KB.

![Diagram of TakeCourse and NewSemester actions]

Figure 4.15: Description of the actions from the Virtual Counselor. The Pre-Condition attributes for each action will be described later in this section, and are shown as blank in this figure.

As shown in the figure, `TakeCourse` is an action that has one argument, `CourseCode`, and `NewSemester` is another action that has one argument, `SemesterLevel`. The preconditions for these two actions are described separately in their own section below.
Describing the State Features

In this section, we describe how the state features in our Student Advisor application can be described using the constructs from the Planning Ontology.

Recall from our discussion in the previous case study that a state feature represents a specific aspect of the environment’s state. As actions are performed against the environment, the state features change their values to reflect the new states that result from these actions. Not all actions will cause all state features to change their values, however. Each feature, in fact, only responds to a certain conditions, and remains the same in all other circumstances. As an example, the \( \text{AtSemesterLevel} \) feature, which represents the semester level the student is at, only response to (i.e., is affected by) the action \( \text{newSemester} \) being performed. All other actions (e.g., \( \text{takeCourse} \), or any other actions had they existed), do not effect the value of this feature.

In our framework, we describe this dynamic (i.e, how a state feature changes its value in response to actions being performed) via two logical expressions.

- **InvariantCond** – This expression specifies the circumstances in which the feature value will remain invariant (not affected by the actions performed).

- **DefiniteValueCond** – This expression specifies the circumstances in which the feature value will take on a definite value.

As the first example, Figure 4.16 shows how the dynamic of the \( \text{AtSemesterLevel}(\text{Level}, S) \) state feature is described in the Course Selection KB.
In the figure, the `DefiniteValueCond` expression specifies that, if the last action is `NewSemester(Level)`, then `AtSemesterLevel` will take on the value of `Level`. Also, the `InvariantCond` expression specifies that if the last action performed is not `newSemester` (with any argument value), then `AtSemesterLevel` will remain the same and keep its previous value.

Similarly, Figure 4.17 shows how the `AtBeginningOfSemester()` state feature is described in the Course Selection KB.

As shown in the figure, `AtBeginningOfSemester()` is `true` if the last action is `newSemester`. If the last action is neither `newSemester` nor `takeCourse`, then `AtBeginningOfSemester()` remains unchanged. It is perhaps useful to note that while the invariant condition for this feature might look a bit redundant, as it excludes the possibility for `AtBeginningOfSemester()` to remain
Figure 4.17: *Ontological description of the AtBeginningOfSemester() state feature.*

Invariant, this condition is valid and will come in handy should the developer decides to extend the system with additional action, such as *(takeBreak or getCoffee, etc.)*.

Progressing to a more complex state feature, Figure 4.18 shows how the dynamics of the *CreditCount()* state feature is specified in the Course Selection KB.

In the figure, if the last action performed is *takeCourse*, then *CreditCount* will take on the new value equal to the sum of its previous value and the credit weight of the course. Otherwise, *CreditCount* will remain unchanged.

Lastly, Figure 4.19 shows the specification of the *HasCompletedBy(Course, Level, S)* state feature. This feature involves
three arguments and is perhaps the most interesting state feature of the application.

As can be seen, the CompletedBy(CourseCode,SemLevel,S) state feature will remain unchanged if either the last action performed is not TakeCourse, or if the last action is TakeCourse but the course taken is not the same as the course code specified in the feature argument. If the last action performed is TakeCourse, where course code is the same as the feature argument, then the feature is true if the semester level at which the course is taken is less then the value of the SemesterLevel argument (i.e., a course won’t be considered
Figure 4.19: *Ontological description of the HasCompletedBy() state feature.*

completed until after the semester in which it was taken).

**Describing Action Preconditions**

With all the relations (ground facts) and state features described, we can now move on to discuss how the action pre-conditions can be described. Figure 4.20 shows the description for *TakeCourse*’s precondition.
As can be seen, the precondition for $\text{TakeCourse}$ is a conjunction of several conditions: The course to be taken must be offered in the current semester; The student has not taken or selected the course before; The workload does not exceed the maximum allowable value (2.5 in this case); and, The student has all the prerequisites for the course. We explain the prerequisite condition by recalling the definition of the $\text{PreReq}$ ground fact above.
The PreReq ground fact defined a binary relations that relates course codes to their prerequisites. These prerequisites took the form of a nested collection (i.e., collection of collections). The outer collections represent the conjunctive prerequisites, and the inner collections represent the disjunctive (alternative) list of courses. The prerequisite for CIS*3700, for example, is that the student must have completed both CIS*2460 and one of CIS*3750 or CIS*3760. This prerequisite is represented by the relational instance PreReq(CIS * 3700, [[CIS * 2460], [CIS * 3750, CIS * 3760]]). As such, in order to check that the student has all the prerequisites, we have to ensure that he or she has completed at least one course in each disjunctive list. As shown in the figure, this check is done via the use of two built-in constructs provided by the Planning Ontology. The ConjunctiveCheck construct represents a conjunction over the members of a collection, while the DisjunctiveCheck construct represents a disjunction over the members of a disjunction. In particular, the HasCompleted expression is evaluated against each the courses from the inner collection to ensure that the student has completed at least one of them.

Similarly, Figure 4.21 shows the precondition for the NewSemester action.

As can be seen, the precondition for NewSemester is a conjunction of several conditions. First, the semester level has to be consecutive (i.e., exactly one greater than the current semester level). Second, the workload of the current semester must be higher than the minimum allowable workload (2 in this case). Third, all the required course for the current semester has been selected. Again, the ConjunctiveCheck construct was used to represent a conjunction over all required courses.
Figure 4.21: PreCondition expression for NewSemester.

Planning Heuristics

Figure 4.22 shows how the example planning heuristic described in our problem formulation section (i.e., “Whenever the VC found itself at the beginning of a semester, then it should immediately select all the courses that are required for that semester before considering any other courses.”) can be described using the Advice concept provided by the Planning Ontology.

In the figure, the TriggeringCondition is an object of type LogicalExpression which states that the planner must be at the beginning
Figure 4.22: An Advice object representing the example planning heuristic from the Student Advisor program.

of a semester. The ActionTemplate is an object of type PartialProgram which is a Sequence of a Search and an Iteration. The search helps ground the Courses variable to the list of courses that are required for the current semester \( L \), and the iteration iterates through these courses and to select them one at a time.
4.2.5 Capturing user’s objectives

With the domain knowledge described in the Course Ontology and the planning problem that underlies course selection described in the Course Selection KB, the last step in building our Virtual Counselor is to design the Objectives Ontology, shown in purple in Figure 4.11. This ontology provides the users (i.e., students) with the necessary language by which to express their objectives, and contains only one concept, ObjectiveStatement. As we have discussed in Section 3.6.3, the ObjectiveStatement’s Overrides property can be populated with RelationalConstraint objects to override the default state feature values, while the GoalExpression can be populated with an object of type LogicalExpression representing the logical description of the goal state.

![Diagram of ObjectiveStatement](image)

Figure 4.23: An example objective statement. An null/blank value for the Overrides property means the user wants to start from the default situation (i.e., SemesterLevel = 0, Semester = fall, CreditCount = 0).

Figure 4.23 show how an example objective, “I would like to complete
CIS*3120 (Digital Systems) before the 6th semester”, can be captured using the ObjectiveStatement concept. In this statement, the Overrides property is left as Null. This fact signifies that the user wants to start from the default situation (i.e., he or she is at semester level 0, in the fall semester, and has no prior credit). The goal state expression, on the other hand, specifies that the user wants a study plan that will result in him or her completed at least 20 credits, in which CIS*3120 is completed before the 6th semester. As we will see in later, this statement, together with the actions’ preconditions and the logical description of the feature variables, will allow the planning framework to figure out a valid study plan that satisfies both the user’s stated objective and the basic requirements such as core courses, prerequisites, and max/min workload.

4.2.6 Translating the problem description

In Section 3.6 we discussed the design of the Translation Engine, and how it translates the various constructs in the Planning Ontology into corresponding Prolog statements. In this section, we shows how the main components of the Student Advisor problem description are translated into Prolog rules. The complete generated Prolog program is included in Appendix B.

Ground Facts

The two ground fact descriptions we discussed above are translated into the following Prolog assertions.

course(cis1000, introduction_to_computer_applications, 0.5).
course(cis1200, introduction_to_computing, 0.5).
course(cis1250, software_design_i, 0.5).
course(cis1910, discrete_structures_in_computing_i, 0.5).
course(cis2030, application_of_microcomputers, 0.5).
... (lines truncated) ...

prereq(cis2030, [[cis1910], [cis2500]]).
prereq(cis2170, [[cis1200, cis1500]]).
prereq(cis2250, [[cis1250], [cis1500]]).
prereq(cis2430, [[cis2500]]).
prereq(cis2460, [[cis2500]]).
prereq(cis2500, [[cis1500]]).
prereq(cis2520, [[cis2500], [cis1910]]).
... (lines truncated) ...

Actions

The 2 actions are translated into the following Prolog fact assertions:

action(takeCourse(CourseCode)).
action(newSemester(SemesterLevel)).

These assertions tell the planning engine that it can select from two different sets of actions when putting together a course taking plan for the student.

State Features

The state features are translated into the following Prolog rules:
atSemesterLevel(L, [A|S]):-  
    A = newSemester(L);  
    not A = newSemester(_, atSemesterLevel(L, S).

atBeginningOfSemester([A|S]):-  
    A = newSemester(_);  
    not A = takeCourse(_, atBeginningOfSemester(S).

creditCount(Count, [A|S]):-  
    A = takeCourse(CourseCode),  
    course(CourseCode, _, Credit),  
    creditCount(C, S), Count is Credit + C;  
    not A = takeCourse(CourseCode),  
    creditCount(Count, Area, S).

hasCompletedBy(Course, Level, [A|S]):-  
    A = take(Course), atSemesterLevel(L, [A|S]), L < Level;  
    not A = take(Course),  
    hasCompletedBy(Course, Level, S).

In the code listing, the commas (“,“) represent logical conjunction and can be read as “AND”, the semicolons (“;”) represent logical disjunction, and can be read as “OR”, and the “:-” symbols represent logical implication and can be read as “IF”. Also, the square brackets ([ ]) represent lists in Prolog, and [A|S] represents the situation (i.e., list of actions) in which A was the last action in the sequence.
The rule for `atSemesterLevel()` is defined as follows: The student is at semester level \( L \) in a given situation if either: a) The last action performed was `NewSemester(L)`, or b) He or she was in semester level \( L \) in the previous situation, and the last action was not a `NewSemester(_)`.

(The underscore symbol represents an anonymous variable in Prolog, which are also known as “don’t care” variable).

**Action PreConditions**

The action preconditions are translated into the following Prolog rules:

```prolog
poss(take(Course), S) :-
    isOfferedIn(Course, Semester),
    inSemester(Semester, S),
    not hasSelected(Course, S),
    hasPreRequisites(Course, S),
    workload(Load, S), Load =< 2.
```

```prolog
poss(newSemester(Level), S) :-
    atSemesterLevel(L, S),
    L < 9, Level is L + 1,
    requiredCourses(L, Courses),
    hasSelectedAll(Course, S),
    workload(Load, S),
    inSemester(Semester, S), ( Semester = summer, Load >= 0; not Semester = summer, Load > 2
```

```
136
```
As we have discussed in previous section, these rules state that it is possible for the student to take a course if a) it is being offered in the current semester, b) it has not been selected/taken before, c) the student has all the prerequisites, and c) the workload doesn’t exceed the maximum allowable workload. Also, starting a new semester (and hence concluding the current semester) is possible if a) the student has not exceed the maximum time limit, b) all required courses for the current semester has been selected, c) the minimum workload is satisfied.

Heuristic Advices

As we have discussed in Section 3.6.2, heuristic advices are translated into Prolog assertions of the form

```
advice($Name, $TriggerCondition, $ActionTemplate)
```

where $Name is the name of the advice, $TriggerCondition is a LogicalExpression object specifying the logical condition under which the advice is applicable, and $ActionTemplate is a PartialProgram object representing a search template. The example heuristic advice we discussed earlier, (i.e., “For each semester, all the required courses should be selected before other courses should be considered”) is translated into the following Prolog assertion:

```
advice(a1,
        atBeginningOfSemester(S),
```
search(
    atSemesterLevel(L, S) &
    requiredCourses(L, Courses)
) :
    foreach(Course, Courses, take(Course))
).

In the listing, the triggering condition is that the student is at the beginning of a semester, and the search template is a Sequence of a search, which helps ground the Courses variable to the list of required courses, and a foreach operation that iterates through each of the required courses and select them one by one. As a quick refresher, the symbols &, is a operator understood by the SitCalc planner as logical symbol for logical conjunction, while the symbols : and foreach, which appear in the action template, are operators understood by the SitCalc planner as sequential composition and iteration, respectively.

4.2.7 Executing User’s Requests

As we have discussed in Section 3.6.3, the system services user’s planning request by first translating the input ObjectiveStatement object, which might contains some RelationalConstraints as overrides and LogicalExpression describing the goal state, into a set of Prolog statements. The example objective shown in Figure 4.23 i.e. “I would like to complete CIS*3120 (Digital Systems) before the 6th semester”, for example, is translated into:

```
goal(S):-
    creditCount(all, Count, S),
```
Count $\geq 20,$

hasCompletedBy(cis3120, 6, S).

In this listing, the goal state is described to be a situation $S$ in which the total credit count is at least 20 (40 courses), and CIS*3120 has been completed by the 6th semester.

Once the input objective have been translated, they are merged into the main knowledge base along with the Actions, PreConditions, State Features. The merged KB are then loaded and executed on a SitCalc planner (e.g., the Golog interpreter) to obtain a plan and return it to the user. Figure 4.24 shows a screen shot showing the execution of the generated program on the Golog planner, a standard Prolog-based SitCalc planner.

Figure 4.24: Executing the generated rule-based program on Golog, a standard SitCalc planner.
4.3 Chapter Summary and Discussion

As we have mentioned in Section 3.3, the primary research objective of this thesis is to prove the feasibility, effectiveness and reusability of our proposed translation approach to integrating planning into ontology-driven applications. In this chapter, we presented two different case studies that we have conducted for this purpose. These two case studies helped demonstrate several important points.

First, they showed that a translation approach to integrating planning into ontology-driven applications is feasible by overcoming both the representability and translatability challenges discussed in Section 3.2. In particular, these two case studies showed that:

- Using an appropriate set of ontological concepts (i.e. the Planning Ontology), planning problems that underlie intelligent ontology-driven applications can be successfully described in an ontology, even if this description involves logical expressions that would otherwise lie beyond the expressibility boundary of ontology and description logics.

- Furthermore, by carefully controlling the vocabulary in the Planning Ontology, we can transparently and non-intrusively control the expressibility of the resulting “language” to ensure that all ontological planning problem descriptions made by the users of the framework are translatable into equivalent Horn clauses. In Section 3.7.1 we showed that through the Planning Ontology, the application developer is automatically restrained to ontological models that are translatable. As can be seen in the case studies, this restraining was intuitive and almost transparent
to the application developer – He or she didn’t have to follow any specific rules in order to stay within the boundary of translatable models. All he or she needed to do was to use the existing vocabulary to intuitively express his or her knowledge, and the framework took care of all boundaries issues automatically.

Second, these two case studies helped demonstrate the generality and reusability of our framework. Despite the fact that the VC and the Trip Planner are two different applications from two different domains with two different sets of requirements and constraints, we were able to apply the framework to build both of them without any modifications. This applicability across applications and domains was possible by the generality of our framework, i.e. the planning constructs from the Planning Ontology in particular.

Third, the case studies helped demonstrate our framework’s effectiveness in coping with complex, real-world planning problem because of its ability to accept heuristic advice the system designer, and hence its ability to effectively cope with the planning complexity of real-world planning problems.
(This page was intentionally left blank)
Chapter 5

Conclusions

In this thesis, we proposed a new translation-based approach for integrating planning into ontology-driven applications, and presented a generic (i.e., domain and application independent) integration framework that allows designers of intelligent systems to incorporate planning capabilities into their ontology-driven applications in a reusable and inter-operable way.

The rationale for our proposed approach is that it offers several important advantages over existing approaches. First, it is more practical and “safer” than language-based approaches, because it allows planning to be integrated into ontology-driven applications using existing, mature and proven tools and languages. Second, it is also more reusable and inter-operable than parallel modeling or ad hoc approaches, because the planning knowledge bases expressed as an ontology are more likely to be understood, and hence reusable, by future applications than a KB expressed in a framework-specific language.

Pursuing this idea required us to overcome two important challenges. First, we needed to ensure that we can represent planning problems in ontology. In
particular, we needed to find a way to represent n-ary predicates, where n is greater than 2, in Description Logics. Second, we needed to ensure that planning problems descriptions can be translated into an equivalent rule-based program. In particular, we needed to find an effective and non-intrusive way of constraining the system designers to making only ontological expressions that can be translated into rule-like expressions. As discussed in Chapter 3, we overcame these two challenges using a simple yet important intuition that an ontology can be thought of, and used, as a language. In particular, the expressibility issue was addressed using a set of ontological constructs that represents logical operators and expressions, while the constraintability issue was addressed by controlling the list of language constructs and their combinations.

We proved the translatability of our language using an induction, and demonstrated the feasibility, reusability and effectiveness of our framework using two independent case studies. In the first case study, we built an intelligent Trip Planner and demonstrated that the representability and translatability challenges can indeed be overcome, and hence the feasibility of our proposed approach. In the second case study, we applied our framework to build an intelligent Student Advisor and demonstrated not only the framework’s effectiveness in coping with complexity in real-world planning problems, but also its generality and applicability across applications and domains.

As we have discussed in Chapter 3, the primary strengths of our proposed framework and its architecture are:

- *Separation of Concerns* – Because planning-related knowledge are kept separated from the domain ontologies and KBs, these KBs can be de-
Veloped and maintained independently from the planning application itself. This independence 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.

- **Framework Independence** – 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.

- **Mature and Proven Formalisms** – 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 planning capability in ontology-driven applications.

- **Transparency and Non-Intrusiveness** – Using our framework, the ontology-driven application designer can continue to think and work in the ontological modeling 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 describe his planning problem in his familiar ontology editing environment, and have the framework handles all the mappings for him.

This thesis has two primary contributions. The first is that it has produced
a practical, effective and reusable integration framework for bringing planning capabilities into ontology-driven applications in a more modular and reusable way [86][87]. This framework consists of the following components:

- A generic and modular architecture for building planning-based ontology-driven intelligent systems.

- A reusable Planning Ontology that can be used to described planning problems that underlie many intelligent systems in an application and domain-independent way. This Planning Ontology also has the potential to serve as a common abstraction language in which planning knowledge and strategies can be encoded in a framework-independent and sharable way.

- A translation engine that translates ontological planning descriptions into executable SitCalc knowledge bases.

The second primary contribution of this thesis is that it also serves as a concrete example on how an ontology could be effectively used as a language, as well as the potential benefits of such usage. As we have discussed in Chapters 3 and 4, viewing and using the Planning Ontology as a language allowed us to accomplish a rather interesting objective – Allowing the system designers to go beyond the the expressivity of ontology (i.e., Description Logics) to describe their planning problems, yet at the same time constraining them to not go beyond the boundary of translatable models. From a language point of view, this is a expressivity problem – Accomplishing this objective required us to be able to effectively control the expressivity of the resulting language. It
must be expressive enough to describe planning problems, yet can not be too expressive that it can not be translated into executable rule-based programs. As such, the success of this research also demonstrates some additional points that are very interesting:

- An ontology can be viewed and used as an mean to create a new language.

- The expressivity of the created language can be effectively controlled. (i.e., By controlling which concepts are included in the ontology, and how these concepts can be combined to form expressions)

- Enforcing the expressivity boundary of the language can be done in a way that is transparent and non-intrusive to the language users. (That is, the user doesn’t have to follow any specific rules in order to stay within the boundary of the language. All he or she needed to do was to use the existing vocabulary to intuitively express his or her knowledge, and the resulting expressions are guaranteed to be within the language boundary.)

Because of these benefits, the “ontology-as-a-language” intuition employed in this thesis could have several potential uses beyond the integration of planning and ontological modeling. One such potential use could be found in the areas of model-driven software development (MDD) [13]. In MDD, the primary goal is to allow software developers to create their software by composing high-level models instead of writing codes in implementation languages such as Java or C++. The rationale for this approach is that by allowing the software developer to work at a higher level of abstraction, software can be produced
and maintained in a much more economical way [99]. One of the most important focuses of current MDD research is in finding an effective way to easily and quickly creating new, application and domain-specific modeling languages that would allow developers to describe their domain-specific knowledge in a rich and precise way. Given the above advantages that ontology-based languages can offer, MDD could be a very promising research area to employ this technique [84].

As for future works, we aim to further refine the framework and apply it to larger and more complex real-life problems. In particular, we aim to provide better tooling support for the users of the framework. Currently, debugging of generated code requires the user to understand SitCalc and how the GOLOG interpreter works, but with better tooling support, this issue can be alleviated. We also seek to further pursue the idea of using ontology as a language in other application domains such as MDD and ODSC to confirm its usefulness.
Bibliography


[5] Franz Baader, Maja Milicic, Carsten Lutz, Ulrike Sattler, and Frank Wolter. Integrating description logics and action formalisms: First re-


155


Software Engineering and the Semantic Web.


(This page was intentionally left blank)
Appendix A

Translated Trip Planning Program

% Ground Facts
%---------------------------------------------------------------

flight(ac123, toronto, paris).
flight(ac223, paris, toronto).
flight(aa100, new_york, london).
flight(aa200, london, new_york).
flight(af100, paris, nantes).
flight(af200, nantes, paris).

train(rf10, nantes, paris).
train(rf20, paris, nantes).

bus(go100, guelph, toronto).
bus(go200, toronto, guelph).
bus(pt100, paris, nantes).
bus(pt200, nantes, paris).

hubAirport(toronto).
hubAirport(paris).

% Actions
%-----------------------------------------

action(takeFlight(FlightNo, Src, Dst)).
action(takeTrain(TrainNo, Src, Dst)).
action(takeBus(BusNo, Src, Dst)).
action(takeRest).
action(getCoffee).
action(drinkCoffee).

groundAction(A):-
A = flight(X, Y, Z), takeFlight(X, Y, Z);
A = train(X, Y, Z), takeTrain(X, Y, Z);
A = bus(X, Y, Z), takeBus(X, Y, Z);
A = takeRest;
A = getCoffee;
A = drinkCoffee.
% State Features
%---------------------------------------------------------------------

stateFeature(hasCoffee(S)).
stateFeature(isRested(S)).
stateFeature(inCity(City, S)).

hasCoffee([A|S]):-  
A = getCoffee;
hasCoffee(S),
not A = drinkCoffee.

isRested([A|S]):-  
A = takeRest;
isRested(S),
(  
not A = takeFlight(_, _, _),
not A = takeTrain(_, _, _)
).  

isRested([]).

inCity(City, [A|S]):-  
A = takeFlight(_, _, City);
A = takeTrain(_, _, City);
A = takeBus(_, _, City);
inCity(City, S),
(
    not A = takeFlight(_, City, _),
    not A = takeTrain(_, City, _),
    not A = takeBus(_, City, _)
).

inCity(guelph, []).

% Pre-Conditions
%-----------------------------------------

poss(getCoffee, S).

poss(drinkCoffee, S):-
    hasCoffee(S).

poss(takeRest, S).

poss(takeBus(_, Src, _), S):-
inCity(Src, S).
poss(takeTrain(_, Src, _), S):-
inCity(Src, S), isRested(S).

poss(takeFlight(_, Src, _), S):-
inCity(Src, S), isRested(S).

% Goal State Description
%-----------------------------------------
goal(S):-
inCity(paris, S), isRested(S), hasCoffee(S).

% Heuristic Advices
%-----------------------------------------
advice(
a1,
inCity(City, S) & destCity(City),
takeBreak
).

advice(
a2,
inCity(City, S) & hubAirport(City),
(
  ground(destCity(Dest)) : flight(_, City, Dest)
  #
  ground(hubAirport(Airport)) : flight(_, City, Airport)
)
).

adviceList([a1, a2]).
Appendix B

Translated Student Advisor Program

%------------------------------------------------------------
% Fact Assertions
%------------------------------------------------------------
semester(fall).
semester(winter).
semester(summer).

semesterLevel(1).
semesterLevel(2).
semesterLevel(3).
semesterLevel(4).
semesterLevel(5).
semesterLevel(6).
semesterLevel(7).
semesterLevel(8).
semesterLevel(9).
semesterLevel(10).
semesterLevel(11).
semesterLevel(12).
semesterLevel(13).
semesterLevel(14).
semesterLevel(15).

course(cis1000, cis, 1000, introduction_to_computer_applications).
course(cis1200, cis, 1200, introduction_to_computing).
course(cis1250, cis, 1250, software_design_i).
course(cis1500, cis, 1500, introduction_to_programming).
course(cis1910, cis, 1910, discrete_structures_in_computing_i).
course(cis2030, cis, 2030, structure_and_application_of_microcomputers).
course(cis2050, cis, 2050, computers_and_society).
course(cis2170, cis, 2170, user_interface_design).
course(cis2250, cis, 2250, software_design_ii).
course(cis2430, cis, 2430, object_oriented_programming).
course(cis2500, cis, 2500, intermediate_programming).
course(cis2520, cis, 2520, data_structures).
course(cis2750, cis, 2750, software_systems_development_and_integration).
course(cis2910, cis, 2910, discrete_structures_in_computing_ii).
course(cis3000, cis, 3000, social_implications_of_computing).
course(cis3090, cis, 3090, parallel_programming).
course(cis3110, cis, 3110, operating_systems).
course(cis3120, cis, 3120, digital_systems).
course(cis3150, cis, 3150, theory_of_computation).
course(cis3190, cis, 3190, software_for_legacy_systems).
course(cis3210, cis, 3210, computer_networks).
course(cis3250, cis, 3250, software_design_iii).
course(cis3260, cis, 3260, software_design_iv).
course(cis3490, cis, 3490, the_analysis_and_design_of_computer_algorithms).
course(cis3530, cis, 3530, data_base_systems_and_concepts).
course(cis3700, cis, 3700, introduction_to_intelligent_systems).
course(cis3750, cis, 3750, system_analysis_and_design_in_applications).
course(cis3760, cis, 3760, software_engineering).
course(cis4000, cis, 4000, applications_of_computing_seminar).
course(cis4050, cis, 4050, advanced_computer_architectures).
course(cis4110, cis, 4110, computer_security).
course(cis4150, cis, 4150, software_reliability_and_testing).
course(cis4210, cis, 4210, telecommunications).
course(cis4250, cis, 4250, software_design_v).
course(cis4300, cis, 4300, human_computer_interaction).
course(cis4410, cis, 4410, trends_in_distributed_systems).
course(cis4430, cis, 4430, information_organization_and_retrieval).
course(cis4450, cis, 4450, special_topics_in_information_science).
course(cis4500, cis, 4500, special_topics_in_computing_science).
course(cis4650, cis, 4650, compilers).
course(cis4720, cis, 4720, image_processing_and_vision).
course(cis4780, cis, 4780, computational_intelligence).
course(cis4800, cis, 4800, computer_graphics).
course(cis4820, cis, 4820, game_programming).
course(cis4900, cis, 4900, computer_science_project).
course(cis4910, cis, 4910, computer_science_thesis).

isOfferedIn(cis1000, summer).
isOfferedIn(cis1000, fall).
isOfferedIn(cis1000, winter).
isOfferedIn(cis1200, fall).
isOfferedIn(cis1200, winter).
isOfferedIn(cis1250, fall).
isOfferedIn(cis1500, fall).
isOfferedIn(cis1500, winter).
isOfferedIn(cis1910, winter).
isOfferedIn(cis2030, fall).
isOfferedIn(cis2050, summer).
isOfferedIn(cis2170, winter).
isOfferedIn(cis2250, winter).
isOfferedIn(cis2430, fall).
isOfferedIn(cis2460, fall).
isOfferedIn(cis2500, winter).
isOfferedIn(cis2520, fall).
isOfferedIn(cis2750, winter).
isOfferedIn(cis2910, fall).
isOfferedIn(cis3000, fall).
isOfferedIn(cis3090, fall).
isOfferedIn(cis3110, winter).
isOfferedIn(cis3120, winter).
isOfferedIn(cis3150, fall).
isOfferedIn(cis3190, winter).
isOfferedIn(cis3210, fall).
isOfferedIn(cis3250, fall).
isOfferedIn(cis3260, fall).
isOfferedIn(cis3490, winter).
isOfferedIn(cis3530, fall).
isOfferedIn(cis3700, winter).
isOfferedIn(cis3750, fall).
isOfferedIn(cis3760, winter).
isOfferedIn(cis4000, fall).
isOfferedIn(cis4000, winter).
isOfferedIn(cis4050, fall).
isOfferedIn(cis4110, winter).
isOfferedIn(cis4150, fall).
isOfferedIn(cis4210, winter).
isOfferedIn(cis4250, fall).
isOfferedIn(cis4300, fall).
isOfferedIn(cis4410, winter).
isOfferedIn(cis4430, winter).
isOfferedIn(cis4650, winter).
isOfferedIn(cis4720, winter).
isOfferedIn(cis4780, fall).
isOfferedIn(cis4800, winter).
isOfferedIn(cis4820, winter).
isOfferedIn(cis4900, summer).
isOfferedIn(cis4900, fall).
isOfferedIn(cis4900, winter).
isOfferedIn(cis4910, summer).
isOfferedIn(cis4910, fall).
isOfferedIn(cis4910, winter).

hasCredit(cis1000, 0.50).
hasCredit(cis1200, 0.50).
hasCredit(cis1250, 0.50).
hasCredit(cis1500, 0.50).
hasCredit(cis1910, 0.50).
hasCredit(cis2030, 0.50).
hasCredit(cis2050, 0.50).
hasCredit(cis2170, 0.75).
hasCredit(cis2250, 0.50).
hasCredit(cis2430, 0.50).
hasCredit(cis2460, 0.50).
hasCredit(cis2500, 0.50).
hasCredit(cis2520, 0.50).
hasCredit(cis2750, 0.75).
hasCredit(cis2910, 0.50).
hasCredit(cis3000, 0.50).
hasCredit(cis3090, 0.50).
hasCredit(cis3110, 0.50).
hasCredit(cis3120, 0.50).
hasCredit(cis3150, 0.50).
hasCredit(cis3190, 0.50).
hasCredit(cis3210, 0.50).
hasCredit(cis3250, 0.50).
hasCredit(cis3260, 0.50).
hasCredit(cis3490, 0.50).
hasCredit(cis3530, 0.50).
hasCredit(cis3700, 0.50).
hasCredit(cis3750, 0.75).
hasCredit(cis3760, 0.75).
hasCredit(cis4000, 0.50).
hasCredit(cis4050, 0.50).
hasCredit(cis4110, 0.50).
hasCredit(cis4150, 0.50).
hasCredit(cis4210, 0.50).
hasCredit(cis4250, 0.50).
hasCredit(cis4300, 0.50).
hasCredit(cis4410, 0.50).
hasCredit(cis4430, 0.50).
hasCredit(cis4450, 0.50).
hasCredit(cis4500, 0.50).
hasCredit(cis4650, 0.50).
hasCredit(cis4720, 0.50).
hasCredit(cis4780, 0.50).
hasCredit(cis4800, 0.50).
hasCredit(cis4820, 0.50).
hasCredit(cis4900, 0.50).
hasCredit(cis4910, 0.50).

prereq(cis1000, []).
prereq(cis1200, []).
prereq(cis1250, []).
prereq(cis1500, []).
prereq(cis1910, []).
prereq(cis2050, []).
prereq(cis3000, []).
prereq(cis4450, []).
prereq(cis4500, []).
prereq(cis4500, []).
prereq(cis3190, []).
prereq(cis4000, []).
prereq(cis4900, []).
prereq(cis2030, [[cis1910], [cis2500]]).
prereq(cis2170, [[cis1200, cis1500]]).
prereq(cis2250, [[cis1250], [cis1500]])
prereq(cis2430, [[cis2500]])
prereq(cis2460, [[cis2500]])
prereq(cis2500, [[cis1500]])
prereq(cis2520, [[cis2500], [cis1910]])
prereq(cis2750, [[cis2430], [cis2520]])
prereq(cis2910, [[cis1500], [cis1910]])
prereq(cis3090, [[cis2030], [cis3110]])
prereq(cis3110, [[cis2500]])
prereq(cis3120, [[cis2030]])
prereq(cis3150, [[cis2750], [cis3490]])
prereq(cis3210, [[cis3110]])
prereq(cis3250, [[cis2250], [cis2500]])
prereq(cis3260, [[cis2430], [cis2750], [cis3250]])
prereq(cis3490, [[cis2910], [cis2520]])
prereq(cis3530, [[cis2520], [cis2750]])
prereq(cis3700, [[cis3750, cis3760], [cis2460]])
prereq(cis3750, [[cis2750]])
prereq(cis3760, [[cis2750]])
prereq(cis4050, [[cis2030], [cis3110], [cis3120]])
prereq(cis4110, [[cis3110]])
prereq(cis4150, [[cis3750, cis3760]])
prereq(cis4210, [[cis3750, cis3760]])
prereq(cis4250, [[cis2750], [cis3260], [cis3750]])
prereq(cis4300, [[cis3110], [cis3750, cis3760]])
prereq(cis4410, [[cis3210], [cis3750, cis3760]]).
prereq(cis4430, [[cis3110], [cis3530], [cis3750, cis3760]]).
prereq(cis4650, [[cis2030], [cis3110], [cis3150]]).
prereq(cis4720, [[cis2750], [cis3110], [cis2460]]).
prereq(cis4780, [[cis3750, cis3760], [cis3490], [cis2460]]).
prereq(cis4800, [[cis3110], [cis3750, cis3760]]).
prereq(cis4820, [[cis3110], [cis3750]]).
prereq(cis4910, [[cis4900]]).

requiredCourses(1, [cis1500]).
requiredCourses(2, [cis1910, cis2500]).
requiredCourses(3, []).
requiredCourses(4, []).
requiredCourses(5, []).
requiredCourses(6, []).
requiredCourses(7, []).
requiredCourses(8, []).
requiredCourses(9, []).
requiredCourses(10, []).
requiredCourses(11, []).
requiredCourses(12, []).
requiredCourses(13, []).
requiredCourses(14, []).
requiredCourses(15, []).
requiredCourses(16, []).
requiredCourses(17, []).
requiredCourses(18, []).
requiredCourses(19, []).
requiredCourses(20, []).

% Actions
% -------------------------------------------------------------
action(take(Course)).
action(newSemester(Level)).

groundAction(A):-
  A = take(Course), course(Course, _, _, _);
  A = newSemester(Level), semesterLevel(Level).

% State Features
% -------------------------------------------------------------
atBeginningOfSemester([A|S]):-
  A = newSemester(_).
atBeginningOfSemester([A|S]):-
  not A = takeCourse(_), atBeginningOfSemester(S).
atBeginningOfSemester([]).

atSemesterLevel(L, [A|S]):-
A = newSemester(L).

atSemesterLevel(L, [A|S]):- 
not A = newSemester(_), atSemesterLevel(L, S).

atSemesterLevel(1, []). 

inSemester(Semester, S):- 
atSemesterLevel(L, S), 
X is (L mod 3),
(  
X = 1, Semester = fall;
X = 2, Semester = winter;
X = 0, Semester = summer
).

workload(Load, [A|S]):- 
A = newSemester(_), Load = 0.
workload(Load, [A|S]):- 
A = take(Course), hasCredit(Course, Credit), 
workload(L, S), Load is L + Credit.
workload(Load, [A|S]):- 
not A = newSemester(_), 
not A = take(_), 
workload(Load, S).
workload(0, []).
creditCount(Count, [A|S]):-
A = take(Course), hasCredit(Course, Credit),
creditCount(C, S), Count is Credit + C;
not A = take(_),
creditCount(Count, S).
creditCount(0, []).

hasSelected(Course, [A|S]):-
A = take(Course);
not A = take(Course), hasSelected(Course, S).

hasSelectedAll(Courses, S):-
Courses = [].
hasSelectedAll([H|T], S):-
hasSelected(H, S), hasSelectedAll(T, S).

hasCompleted(Course, S):-
atSemesterLevel(L, S),
hasCompletedBy(Course, L, S). 

hasCompletedBy(Course, Level, [A|S]):-
A = take(Course), atSemesterLevel(L, [A|S]), L < Level;
not A = take(Course), hasCompletedBy(Course, Level, S).

stateFeature(atSemesterLevel(L, S)).
stateFeature(inSemester(Semester, S)).
stateFeature(atBeginningOfSemester(S)).
stateFeature(workload(Load, S)).
stateFeature(creditCount(Count, S)).
stateFeature(hasSelected(Course, S)).
stateFeature(hasSelectedAll(Courses, S)).
stateFeature(hasCompletedBy(Course, L, S)).
stateFeature(hasCompleted(Course, S)).

%  *-----------------------------------------------------------------------
%  %  PreConditions
%  %  *-----------------------------------------------------------------------

poss(newSemester(Level), S):-
  atSemesterLevel(L, S), %L < 9,
  Level is L + 1,
  requiredCourses(L, Courses), hasSelectedAll(Course, S),
  workload(Load, S),
  inSemester(Semester, S),
  (  
    Semester = summer, Load >= 0;
    not Semester = summer, Load > 2
  ).

poss(take(Course), S):-
  isOfferedIn(Course, Semester),
inSemester(Semester, S),
not hasSelected(Course, S),
prereq(Course, Prereq), satisfy_all(Prereq, S),
workload(Load, S), Load =< 2.

% ------------------------------------------------------------
% Goals
% ------------------------------------------------------------
goal(S):-
    hasCompletedBy(cis3190, 6, S),
creditCount(Count, S), Count >= 20.

% ------------------------------------------------------------
% Advices
% ------------------------------------------------------------
adviceList([a1]).

advice(
    a1,
    atBeginningOfSemester(S)
& atSemesterLevel(L, S)
& requiredCourses(L, Courses)
& ~(Courses = []),
foreach(Course, Courses, take(Course))
).
\texttt{maxDepth(50)}. 