Representing an Agent’s Incomplete Knowledge for Planning

by

Ronald Peter Andrew Petrick

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1998

©Ronald Peter Andrew Petrick 1998
I hereby declare that I am the sole author of this thesis.
I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.
The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.
Abstract

In many domains, an agent may not have complete knowledge of its environment. Before acting to achieve a goal in such a domain, an agent may have to update its model of the world by sensing the real world to gather more information. Manipulating the world through action may also affect the agent’s knowledge. Therefore, being able to represent an agent’s knowledge and update that knowledge as it changes is essential for both generating and executing plans.

During the process of planning, an agent must reason about the effects of the actions needed to achieve a desired goal. However, an agent must make the distinction between reasoning about a sequence of actions at plan time and actually executing a sequence of actions. The complication arises from the fact that the effects of actions at plan time are often quite different from the effects of actions at execution time. Moreover, at plan time the agent must know that the plan will achieve its goals while at execution time the agent must have sufficient knowledge at each step of the plan in order to execute it.

This thesis describes a formalism for modelling an agent’s incomplete knowledge. The standard STRIPS representation is extended to allow an agent’s knowledge to be represented by a collection of databases, with each database storing a specific type of knowledge. This extension is necessary in order to distinguish between the types of knowledge that an agent may gain during the process of planning, compared with the knowledge that it may obtain during execution. The agent’s knowledge state is formally defined by providing a translation of the database contents to formulas of a modal logic. Algorithms are given for updating an agent’s knowledge while preserving the conditions necessary for maintaining a consistent knowledge state. A sound but incomplete inference procedure is described that allows queries of the agent’s knowledge to be made.
in order that knowledge conditions, such as those required to select the appropriate actions during the planning process, can be satisfied. This research also addresses how an agent’s actions can be represented, making explicit the separation between plan time and execution time effects. The result is that an agent’s knowledge can be projected over a sequence of either planned or executed actions. The notion of an exception is developed as a means of managing the interaction between knowledge of different types. Examples are given that illustrate reasoning about sequences of actions both at plan time and at execution time. The examples also demonstrate how the knowledge representation, action representation, inference procedure, and database update rules interact. In developing a STRIPS-like representation for incomplete knowledge that clearly separates plan time and execution time considerations, this thesis provides the foundation for designing planners that have the ability to model knowledge producing actions and build contingent plans based on information that will only become known during execution.
Acknowledgements

First, I would like thank my supervisor, Dr. Fahiem Bacchus, for his guidance during the preparation of this thesis. The ideas, observations, and insights that came out of our many discussions contributed greatly to the completion of this work. Thank you Fahiem!

I am grateful to my readers, Dr. Robin Cohen and Dr. Ken Salem, for their helpful comments and suggestions.

I would like to acknowledge the University of Waterloo, Communications and Information Technology Ontario, and the Natural Sciences and Engineering Research Council for their financial support.

I am also very thankful for all of the friends I made during my time at the University of Waterloo. They’re the ones that made my experiences here so enjoyable and memorable. Thank you all!

Finally, I would like to thank my family, in particular my aunts for being there for me, and my brother Mark for all the long distance phone conversations.

I would especially like to thank my parents for all their love, support, and encouragement during my time at Waterloo. They helped keep me excited and interested in the work I was doing here. This thesis is dedicated to them.
Contents

1 Introduction 1
  1.1 Motivation ................................................. 3
  1.2 Knowledge .................................................. 4
    1.2.1 Classical Planning and Complete Knowledge .......... 5
    1.2.2 Incomplete Knowledge .................................. 8
    1.2.3 Modal Logic of Knowledge ............................. 9
  1.3 The Planning Problem Revisited ............................ 10
  1.4 Organization of the Thesis ................................ 12

2 Background and Related Work 15
  2.1 The Internet Softbot Project .............................. 15
  2.2 Knowledge and Planning ................................... 21
  2.3 STRIPS, ADL, and Database Representation ............... 23
  2.4 Contingency Planning ..................................... 25

3 Representing the Agent’s Knowledge 31
  3.1 Extending the STRIPS Representation ...................... 31
  3.2 Knowledge, Rigid Terms, and Constant Domains .......... 33
  3.3 The Databases .............................................. 36
### 7 Examples

- **7.1 Open Safe Domain** ........................................... 83
- **7.2 Medical Domain** ............................................ 88
- **7.3 UNIX Domain: Filesystem Actions** ......................... 92
- **7.4 UNIX Domain: Network Actions** ........................... 96

### 8 Future Work and Conclusions

- **8.1 Implementation** ............................................. 99
- **8.2 Extensions** .................................................. 101
- **8.3 Contributions and Conclusions** ............................. 103

### Bibliography

107
List of Tables

1.1 Action specifications from the Blocksworld domain ................... 7
4.1 Inference algorithm .......................................................... 50
4.2 Primitive queries .............................................................. 59
5.1 Action specification grammar .............................................. 65
5.2 Database add and delete primitives ..................................... 67
5.3 Action specification for the UNIX command wc ..................... 68
6.1 Exception handling rules ..................................................... 80
7.1 Open safe domain actions ................................................... 84
7.2 Medical domain actions ..................................................... 89
7.3 UNIX domain filesystem actions ........................................ 93
7.4 UNIX domain network actions ............................................. 97
## List of Figures

1.1 An action’s transformation of a world state .......................... 3

5.1 Applying the \textit{wc thesis.tex} action .................................. 69

7.1 Applying the \textit{dialComb(safe, 27-54-13)} action ................. 86

7.2 Plan time effects of \textit{readComb–dialComb} action sequence .... 87

7.3 Execution time effects of \textit{readComb–dialComb} action sequence . 88

7.4 Plan time effects of a conditional action sequence in the medical domain 91

7.5 Plan time effects of a conditional action sequence in the UNIX domain . 95
Chapter 1

Introduction

Many complex tasks require the ability to plan, the ability to reason about the sequence of actions that are needed to achieve a goal in a given situation. This ability is often essential for the successful completion of the task. For instance, consider the following scenarios:

- A person planning a car trip needs to consider tasks such as filling the car with gas, packing the suitcases, putting the luggage in the car, making hotel reservations, and planning the route to take. The order in which these actions are performed is important to the success of the trip (for instance, the car should be packed before starting to drive).

- An office Mail-bot has the responsibility of delivering departmental mail. The robot needs to determine the path it should take in order to correctly deliver its mail while maintaining delivery constraints (such as time constraints on priority messages, or resource constraints on its energy usage). The robot generates a plan to deliver the mail and responds to unforeseen problems such as obstacles in its path or delivering mail to people that have changed offices.
A planning agent in an operating system environment aids novice users by allowing them to specify high level operations to be performed (for instance, “compress all postscript files on the disk”). The actual sequence of operating system commands that need to be executed to perform the high level operation is automatically generated by the agent.

Each of these examples illustrate a planning domain. More formally, a planning domain is specified by (1) a description of the initial world state, (2) a description of the goal that is to be achieved, and (3) a description of the actions that are available in the domain.

To determine what actions to apply to achieve the desired goals, planning systems work with representations that model the real world. By manipulating the representations these systems can determine the sequence of actions that need to be performed. Specifically, the initial information about the world is represented in a formal way. Actions in the domain are modelled by representing their preconditions and effects. The preconditions state the conditions that must hold in the world representation before the action can be applied. An action’s effects modify the representation of the world in such a way that the new world representation corresponds to the configuration of the real world after the action is actually executed.

Figure 1.1 illustrates the correspondence between the real world and the representation of the real world. $W_{\text{initial}}$ indicates the initial state of the real world, while $W_{\text{rep}}$ indicates the corresponding representation. The effects of action $a$ transform $W_{\text{initial}}$ to a new world, $W_{\text{new}}'$. Action $a$’s representation transforms $W_{\text{rep}}$ to a new world representation, $W_{\text{rep}}'$. The action representation must maintain the correspondence between the new state of the real world, $W_{\text{new}}'$, and the new representation of this world state, $W_{\text{rep}}'$.

The process of planning is the generation of a sequence of actions (a plan) that when applied to the initial world state will transform the world so as to bring about the achieve-
1.1. MOTIVATION

Consider again the example of a planning agent interacting in a complex domain such as the UNIX operating system environment. A formal description of the planning problem for this domain could include information about the filesystem (for instance, descriptions of known files including their path, access permissions, and size), formal descriptions of the UNIX actions available to the planner (in this case the lower level UNIX commands such as `ls`, `compress`, `rm`, etc., with a description of their preconditions and effects), and a description of the goal conditions (the high level conditions to be achieved).

Applying planning to this problem has a number of advantages. The possible high-level operations need not be determined a priori. It isn’t necessary to pre-generate a sequence of actions for every possible high-level operation. Instead, the planner is able to figure out a required sequence of actions as needed, from the formal description it has of the world, the available actions, and the goal it is trying to achieve.

\[ W_{\text{initial}} \xrightarrow{\text{Action } a} W'_{\text{new}} \]

\[ W_{\text{rep}} \xrightarrow{\text{Action } a' \text{'s representation}} W'_{\text{rep}} \]

Figure 1.1: An action’s transformation of a world state

1. Such planning agents are a type of software agent or software robot that exists in a software domain and has a planning module as one of its main components. The term planning agent or simply agent will be used interchangeably in this thesis to refer to the planner. See [WJ95] for a more extensive introduction to intelligent agents.
Since the sequence of actions is not predetermined, the planner is able to adapt to unforeseen changes in the environment. For instance, an operating system problem may disrupt services, making some actions unavailable. The planner may still be able to generate a plan that achieves the required goal by using other actions. Conversely, new services may be added. A planner is able to use these new commands as soon as their formal descriptions are made available. Thus, all possible scenarios that could arise do not need to be addressed beforehand. Instead, the planner has the ability to develop plans that can meet the needs of a given situation as the circumstances dictate, and make use of the resources it has available to it.

Users of such systems also benefit from planning technology. A potential user does not have to worry about the “low-level” details necessary to solve a problem, but can instead defer such work to the planning system. This is especially beneficial for novice users who may not be familiar with the domain being used (for instance, a new UNIX user). This also frees users to deal with more abstract concepts while leaving the details to be filled in by the planner.

1.2 Knowledge

When generating a plan, a requirement is that the plan be correct. That is, at each step of the plan, the preconditions of the action at that step should be satisfied, and the overall effects of the plan should produce a world state in which the goal conditions have been satisfied. During the planning process, then, the planner must be able to reason correctly about the preconditions and effects of actions.

But, in order to verify precondition requirements, or reason about an action’s effects on the world state, reasoning about actions involves testing assertions about the world state, or more specifically, what an agent knows about the world state. Thus, the ability
1.2. KNOWLEDGE

to reason about actions requires the ability to reason about knowledge.

Reasoning about knowledge requires that a number of complicated issues be addressed. A fundamental problem is how an agent’s knowledge can be represented. One way to do this is to maintain a database or collection of facts that an agent has about the world. This database serves as a model of the agent’s state of knowledge about the environment.

A planning agent, however, is not just a static observer of the world; it also has the ability to manipulate the world. If an agent’s knowledge of the world is to remain correct, a way is needed to update this knowledge as it changes when actions are executed. This problem is further complicated if an agent also has the ability to sense. Sensing actions typically do not change the world. They simply allow the agent to observe the world and provide a way for it to gather new information and update its world model. Thus, reasoning about knowledge requires a means for modeling how actions affect knowledge.

1.2.1 Classical Planning and Complete Knowledge

In the past, many planners have employed classical planning techniques, characterized by restrictions in the planning domain and the types of problems that can be solved. One important assumption that classical planners make is that the initial world state is completely specified. Also, actions in a classical planning domain are deterministic and map world states to new world states, preserving the completeness of the world description. Thus, a planning agent does not need the ability to sense the environment. Its world description is always complete, and the actions preserve the agent’s complete knowledge of the world. Complete knowledge greatly simplifies the problem of reasoning.

An example of a classical planner is the STRIPS planner [FN71]. The standard STRIPS representation consists of a single database, $D$, that stores a list of the posi-
tive, ground, atomic facts that the planner knows about the world. STRIPS also makes use of a *closed world assumption*: any fact that is not explicitly stored in the database, \( D \), is assumed to be false. By using the closed world assumption, negative facts need not be explicitly stored. For instance, the database \( D = \{ \text{on}(C), \text{on}(B, C), \text{on}(A, C), \text{clear}(A) \} \) could describe the stacking of blocks on a table. Block \( C \) is on the table with block \( B \) stacked on top of it and block \( A \) stacked on \( B \). Since \( A \) is at the top, it is “clear” (there are no blocks stacked on top of it). Facts not in \( D \) such as \( \neg \text{on}(C, A) \) (“block \( C \) is not on block \( A \)”), \( \neg \text{on}(B) \) (“block \( B \) is not on the table”), and \( \neg \text{clear}(C) \) (“block \( C \) is not clear) are implicitly represented.

Actions in STRIPS are represented as parameterized database update rules. An action is described by its parameters, preconditions, add list, and delete list. The parameters provide an action template that can then be instantiated by constants representing particular objects. When instantiated, the parameters in the action specification are replaced by constants to produce ground literals. The preconditions specify a conjunction of literals that must be satisfied for the action to be applied. To determine if the action’s preconditions are satisfied, we check to see if these literals are in \( D \), or for the case of negative literals, that they are not in \( D \). The effects of actions are modelled by two lists that describe the changes the action makes to \( D \). The add list indicates the literals that should be added to \( D \), and the delete list specifies the literals that should be removed from \( D \) as a result of applying the action. Together, the add and delete lists update a database to produce a new database representing the new state of the world.

Consider a planning example from the Blocksworld domain, consisting of a table with three lettered blocks on it. A robotic arm has the ability to pick up a single block and move it onto another block or the table. The arm is only able to hold one block at a time, and only blocks that are “clear” can be picked up. Specifications for two of the robot actions, *pickup* (pick up a block from the table), and *stack* (stack a block on
1.2. KNOWLEDGE

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Add List</th>
<th>Delete List</th>
</tr>
</thead>
<tbody>
<tr>
<td>pickup(?x)</td>
<td>handempty</td>
<td>holding(?x)</td>
<td>handempty</td>
</tr>
<tr>
<td></td>
<td>ontable(?x)</td>
<td>clear(?x)</td>
<td>ontable(?x)</td>
</tr>
<tr>
<td></td>
<td>clear(?x)</td>
<td></td>
<td>clear(?x)</td>
</tr>
<tr>
<td>stack(?x, ?y)</td>
<td>holding(?x)</td>
<td>on(?x, ?y)</td>
<td>holding(?x)</td>
</tr>
<tr>
<td></td>
<td>clear(?y)</td>
<td>clear(?x)</td>
<td>clear(?y)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>handempty</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Action specifications from the Blocksworld domain

Another block) are shown in Table 1.1.

The initial configuration of the world is represented by the list of literals \( D_0 = \{ \text{ontable}(A), \text{ontable}(C), \text{on}(B, C), \text{clear}(A), \text{clear}(B), \text{handempty} \} \), indicating that \( A \) is on the table, \( B \) is stacked on \( C \), and the robot hand is empty. The goal is to stack block \( A \) on block \( B \) and have the robot hand empty. These conditions are described by the goal \( G = \{ \text{on}(A, B), \text{handempty} \} \).

One plan that achieves the goal is the action sequence \( \text{pickup}(A), \text{stack}(A,B) \). Both \text{handempty} and \text{ontable}(A) are in \( D_0 \) so the preconditions of \( \text{pickup}(A) \) are met. Applying the effects of this action produces a new world state \( D_1 = \{ \text{ontable}(C), \text{on}(B, C), \text{clear}(B), \text{holding}(A) \} \), where the literals \text{ontable}(A), \text{clear}(A), and \text{handempty} have been removed from \( D_0 \) and the literal \text{holding}(A) has been added. Now the \text{stack}(A,B) action can be performed since \text{holding}(A) and \text{clear}(B) are in \( D_1 \). The effects of the action are applied, resulting in the literals \text{on}(A,B), \text{clear}(A), and \text{handempty} being added to \( D_1 \), and \text{holding}(A) and \text{clear}(B) being deleted. The result is a new database \( D_2 = \{ \text{on}(A,B), \text{ontable}(C), \text{on}(B, C), \text{clear}(A), \text{handempty} \} \), where the goal \( G \) is satisfied.
1.2.2 Incomplete Knowledge

In many domains, however, the assumption of complete knowledge is not realistic. For instance, an operating system agent in the UNIX environment may not have complete knowledge of all the files in the filesystem, and an Internet agent will not know the contents of all the web pages on the World Wide Web. In such cases an agent must be able to reason with incomplete knowledge.

Incomplete knowledge makes reasoning more complex since the closed world assumption can no longer be applied. If a fact is not present in the agent’s knowledge, it cannot assume that it is false. Consider a STRIPS database, $D = \{ \text{in(luggage, car-trunk)}, \text{road-blocked(Waterloo, Thunder Bay)} \}$, representing the facts that “the luggage is in the trunk of the car” and “the road from Waterloo to Thunder Bay is blocked”. By the closed world assumption, a fact not in $D$ such as $\neg\text{in(spare-tire, car-trunk)}$ (“the spare tire is not in the car trunk”) is also implicitly represented, even though this information may simply be unknown to the agent (and possibly true in the real world). We need to explicitly model the agent’s knowledge, and the fact that this knowledge is incomplete.

One way an agent is able to gather more information about its environment is through sensing. Sensing actions are examples of knowledge-producing actions that return information about the world without necessarily making changes to it. A sensing action that models reading a thermometer may provide the agent with the outside temperature, without changing the temperature. Classical planners, including STRIPS, are not able to deal with incomplete knowledge or knowledge-producing actions. Modelling knowledge-producing actions requires a specification that reflects the changes that such actions make to the agent’s knowledge.
1.2.3 Modal Logic of Knowledge

Standard modal logics of knowledge provide a formal representation of knowledge. Syntactically, the modal operator $K$ is added to an ordinary first-order language, extending it by the rule: if $\phi$ is a formula then so is $K(\phi)$. A formula such as $K(\phi)$ can be interpreted informally as “the agent knows $\phi$”. Semantically, the language is interpreted over a collection of worlds, $W$, where each world $w \in W$ is a first-order model. Worlds are related to each other by an accessibility relation. In this case, the S5 relation is used to indicate that every world is accessible from every other world. A non-modal formula $\phi$ is true at a particular world $w$ ($w \models \phi$) iff it is true according to the standard rules for interpreting first-order formulas. A modal formula of the form $K(\phi)$ is interpreted to be true at $w$ iff $\phi$ is true at every world accessible from $w$. Given that every world is accessible from every other world, $\phi$ must be true at every world in $W$, including $w$.

The worlds in the collection $W$ are known as possible worlds. Besides the true world, there are a number of other possible states or worlds. An agent considers a world possible if it is consistent with everything that the agent knows. Given the current information an agent has about the world, he may not be able to tell which of a number of possible worlds is in fact the real world. For instance, if an agent knows that it is raining in Waterloo, then this fact must be true at all the worlds the agent considers possible. However, the agent may not know if it is raining in Thunder Bay. Thus, in the set of worlds the agent considers possible there may be worlds in which it is raining in Thunder Bay, and worlds in which it is not raining.

Formally, the set $W$ models the agent’s incomplete knowledge. If an agent knows a fact $\phi$, then $\phi$ must be true in every world the agent considers possible. Not knowing whether or not a formula $\phi$ is true means that there will be worlds in $W$ where $\phi$ is true and worlds where $\phi$ is false. The fewer worlds that the agent considers possible, the more
certain he is about the state of the real world.

A language like modal logic provides a means for formally describing an agent’s knowledge. In a representation such as STRIPS where a database, $\mathcal{D}$, is used to represent facts about the world, the contents of the database can be translated to a set of logical formulas in modal logic. If $\mathbf{KB}$ is used to represent the translated set of modal logic formulas, then formally, for any ground atomic formula $\phi$:

if $\phi \in \mathcal{D}$ then $\mathbf{KB}$ contains the formula $K(\phi)$,

if $\phi \notin \mathcal{D}$ then $\mathbf{KB}$ contains the formula $K(\neg \phi)$,

No other formula $\phi$ is in $\mathbf{KB}$.

The agent’s knowledge is formally characterized by this correspondence between database contents and logical formulas. By doing so, the logic’s semantics can then be used as the underlying semantics of the representation. The knowledge state of the agent is simply the conjunction of the formulas in $\mathbf{KB}$.

1.3 The Planning Problem Revisited

This thesis addresses the problem of how agents who must operate in environments with incomplete knowledge can generate and execute plans. In particular, the focus is on representing and updating the kinds of incomplete knowledge that would be useful to a planning agent capable of sensing and manipulating its environment. Moreover, the representation scheme that is developed is oriented towards use in actual planning systems.

The knowledge representation framework focuses on the case where an agent has correct but incomplete knowledge of the environment. It addresses the problem of deciding what restrictions should be placed on the types of knowledge represented, to balance
1.3. THE PLANNING PROBLEM REVISITED

the tradeoffs between expressiveness of the representation and efficiency of reasoning. The representation also provides a means of modelling the effects of sensors and of information that an agent will come to know at some point in the future.

The approach adopted is similar to the traditional STRIPS representation. However, instead of using a single database, a collection of databases is used to model the different types of knowledge an agent has. We formally characterize the agent’s knowledge state by providing a translation from the database contents to a set of logical formulas in modal logic. The result is a subset of first-order logic that is expressive enough to model knowledge in a variety of domains.

The problem of addressing what kinds of inferences an agent with incomplete knowledge can make is also investigated. In particular, some cases in which an agent can reason while having a “localized” form of complete information are studied. The results are formalized in an inference algorithm that is sound but incomplete.

A specification for describing actions is presented, again extending the STRIPS representation. The specification, however, is enhanced to distinguish between the effects of an action at plan time and the effects of an action at execution time. These may be substantially different. For instance, say that an agent in the UNIX domain is considering the command `ls` that has the effect of listing all the files in a directory. At plan time, all the agent will know is that after this action it will know all the files in the directory. However, the actual names of those files will not be known until the action is executed. Actions are also augmented to operate on the extended database representation. This also makes the action specification more powerful in that the changes an action makes to different types of knowledge can be modelled.

The action specification provides a means for updating an agent’s knowledge. The approach of separating plan time effects from execution time effects allows the agent’s knowledge state to be projected through a sequence of planned actions, by applying the
actions’ plan time effects to the agent’s initial knowledge state. The result is a sequence of intermediate knowledge states. This same approach can be applied to the actions’ execution time effects. In this way, the formalism can also support a plan execution module. The separation of plan time from execution time effects is an important theme of the thesis and its investigation helps develop a better understanding of some of the issues involved in managing the tradeoffs between planning and execution.

1.4 Organization of the Thesis

The remainder of the thesis develops the approach to modelling the incomplete knowledge of an agent, for the purpose of constructing effective planning systems. In Chapter 2, background and related work is reviewed. In particular, some of the important projects that have motivated this work are briefly described and contrasted with the work presented here. A framework for representing knowledge is presented in Chapter 3. A formal semantics along with a description of the syntax of the representation are provided. The conditions required for maintaining consistent knowledge states in this representation are also given. An inference procedure is given in Chapter 4 that describes how inferences are made from the represented knowledge. As well, a proof is presented that shows the inference algorithm is sound. Chapter 5 discusses how actions are represented, and how knowledge is updated. A description of the action specification is presented along with examples illustrating how actions operate in conjunction with the inference algorithm and the representational framework. In Chapter 6, the notion of an “exception” is developed to handle the interactions between different types of knowledge. An algorithm for managing simple knowledge exceptions is given. A number of in depth examples are investigated in Chapter 7 to illustrate the power of the representation when reasoning in a planning situation. Examples from the current literature are used
1.4. ORGANIZATION OF THE THESIS

for comparison. Finally, in Chapter 8 the contributions and conclusions of this work are
summarized. This chapter also describes some of the directions that this work could be
taken, and mentions possible extensions and future research.
Chapter 2

Background and Related Work

Recently, the issue of planning with correct but incomplete information has received much attention, as there are many domains that can be reasonably modelled with this characterization. In this chapter, projects that have addressed this important problem and motivated the framework developed in the thesis are compared and contrasted.

2.1 The Internet Softbot Project

Etzioni, Weld, and Golden have pursued ongoing research into designing software agents for use in the UNIX and Internet environments [ELST93, EW94, EW95]. In particular, the University of Washington’s Internet Softbot project has investigated and developed planning technology to address the problem of managing correct, but incomplete knowledge in complex, real-world domains like the Internet. The sheer demand for tools to manage the growth of information in these domains has greatly increased the importance of agents that can operate in such environments [Nwa96].

The Internet Softbot is an Internet information agent,¹ a software agent that acts as

¹See [Nwa96] for a description of the diverse types of agents being investigated.
an intelligent personal digital assistant. The agent operates in a software domain, the UNIX operating system environment, using commands like `ftp` and `lpr` as effectors to manipulate the environment, and commands such as `ls` and `finger` as sensors to gather more information from its surroundings.

The Softbot framework allows a user to express high-level, goal directed requests in a logical language. This logical language is a subset of first-order logic that allows goals to be expressed using conjunctions, disjunctions, negations, and universal and existential quantification. The Softbot dynamically generates a sequence of Internet and UNIX commands by using AI planning techniques to try to achieve the user’s goal. The set of action schemata used by the planner are based on the databases, utilities, software commands, and information sources the agent has access to. The Softbot can execute a sequence of actions, gather more information about the world and retain it for future decisions. Goals that the Softbot can achieve include notification requests, (such as requests to monitor disk utility), constraint enforcement (such as ensuring all files in a directory are readable), object location and manipulation requests (for instance, printing all postscript files), as well as incompletely specified requests (such as printing a file to any free printer).

One of the central components of the Softbot architecture is the XII (eXecution and Incomplete Information) partial order planner [EHW+92, EGW97]. XII is an extension of UCPOP, a partial order planner with an expressive action representation language based on a subset of an extended STRIPS language called ADL [PW92] (see Section 2.3 for further details). Unlike UCPOP however, XII interleaves planning, sensing, and execution to achieve goals in the presence of incomplete information. The planner is both sound (the plan is guaranteed to achieve the goal) and complete (if a plan exists, XII

---

Partial order planning is a method of planning that generates plans by searching through the space of partial plans.
2.1. THE INTERNET SOFTBOT PROJECT

will find it).

XII supports an action representation language, XIIL, that extends UCPOP’s representation language to add support for sensing actions. The language attempts to reconcile the problems faced with representing actions whose primary function is obtaining information rather than changing the physical world and actions that may have both causal and observational effects. To address these problems, XIIL uses annotations to distinguish between causal and observational effects in an attempt to balance expressiveness and efficiency. Action schemata are specified in terms of their parameters, preconditions and effects. Effects are divided into effects that change the physical world (annotated with `cause`) and observational effects that change the planner’s state of information by observing the truth of a proposition or identifying an object that has a particular property (annotated with `observe`). Goals are also divided into causal goals of achievement (annotated with `satisfy`) and observational information gathering goals (annotated with `find-out`).

By focusing on the Internet domain, the problem of modelling some conditions, such as the effects of noisy sensors, is avoided. Instead, the assumption that an agent’s knowledge of the world is correct can be made. However, other problems become more important. For instance, it is not obvious how to satisfy a universal goal such as making all the files in a directory readable, in the presence of incomplete information. Redundant sensing is also a concern; we do not want the planner to insert sensing actions into the plan when the agent already knows the information to be sensed.

Since the Softbot’s knowledge of the Internet is incomplete, it cannot make the closed world assumption. But, there are situations where it is possible to represent the fact that the agent may have complete information on a particular topic. The UNIX command `ls -al` is an example of an action that provides the agent with complete, closed world information about the names, sizes, and permissions of all the files in a particular directory.
This type of *local closed world (LCW)* information gives the Softbot the ability to make more complex inferences in certain circumstances [GEW94].

The agent’s model of the world, $D_M$, is represented as a database of ground literals. Since this model is incomplete and the closed world assumption cannot be used, negative instances are explicitly stored in $D_M$. To represent local closed world information, a meta-level database, $D_C$, is used. LCW information gives the planner the ability to reason that it knows all the objects satisfying a particular formula. For instance, after executing the UNIX command `ls -al /project`, the agent knows the facts `in-dir(makefile, /project)` and `in-dir(source.c, /project)`. These facts are added to $D_M$. However, to store the additional local closed world information that the agent “knows all the files in directory /project” the formula `in-dir(x, /project)` is added to $D_C$. Now the agent can reason about whether the file `thesis.tex` in directory `/project`. First it checks $D_M$. Since `in-dir(thesis.tex, /project)` is not listed in $D_M$, it cannot conclude anything from this database. However, since `in-dir(x, /project)` is in $D_C$, this LCW formula indicates that all known instances that satisfy the formula are stored in $D_M$. Since `in-dir(thesis.tex, /project)` is not one of these instances, the agent can conclude $\neg in-dir(thesis.tex, /project)$. If the formula `in-dir(x, /project)` was not in $D_C$, then the agent would not have LCW information about the files in directory `/project` and would have to conclude that `in-dir(thesis.tex, /project)` is unknown.

The procedure described for querying and updating LCW information is both sound and tractable. Etzioni et al. [EGW94, EGW97] have also provided conservative rules for updating LCW information due to information gain, information loss, or domain changes, and have developed theorems for reasoning with the composition, conjunction, disjunction, and negation of LCW information.

Experiments show that LCW reasoning greatly improves the Softbot’s performance by substantially cutting down on the amount of redundant information gathering. Plans
are often shorter since redundant sensing is removed. When faced with unachievable goals, LCW reasoning allows the agent to fail quickly. The rules for updating LCW information form a sound, linear time algorithm resulting in a small overhead compared to the savings in performance and reduction in redundant sensing.

The Internet Softbot research has shown that the vast size and dynamic nature of the Internet requires that planners be efficient, and as a result, that better algorithms are needed for planning with incomplete information. Work has also been done with a different action representation language called SADL (Sensory Action Description Language) that attempts to cover the middle ground between expressiveness and efficiency. SADL uses a different set of annotations that introduce a temporal element to the representation [GW96].

This research has also illustrated some of the basic advantages that planning has in such domains. Planning has the ability to avoid the rigid control flow that scripts or programs have. Planners can also automatically decompose complex goal expressions so as to solve them with divide and conquer techniques. In a domain such as the Internet, the Softbot is able to integrate resources from multiple independent sources, and can be easily expanded when a new facility is available without reprogramming the planner.

This thesis shares a similar motivation to the Internet Softbot work. The goal of providing agents with planning capabilities to manage the problem of handling incomplete knowledge and knowledge-producing actions is the primary focus of this thesis. In particular, the Internet Softbot work on local closed worlds has influenced some of the ideas presented here. This thesis also focuses on examples taken from the UNIX and Internet domains. However, there are some significant differences between this thesis and the Softbot research.

First, much of their approach seems to be tied to a particular planning technology, partial order planning [GEW96]. This often has the unfortunate effect of making the se-
mantics of their representations and algorithms much more difficult to understand. The approach presented here attempts to provide a clearer separation between the issues that involve the semantics of actions and the agent’s knowledge, and the semantics of planning and the execution of plans.

Second, their work is also very closely tied to the effects of actions that occur at execution time. For instance, the algorithms they develop for reasoning about locally closed world conditions assume that the actions achieving such conditions have been executed. As a result, the planning system they construct is forced to interleave planning and execution in an inflexible manner. In particular, their system does not deal well with domains where executing actions can produce irreversible changes. As well, at times the separation between planning and execution is often difficult to understand, as is how the interleaving is actually achieved in practice.

Third, this thesis extends and also redefines some of the LCW notions introduced in the Softbot research. The LCW mechanism in the Softbot work is awkward for handling some types of knowledge such as the uniqueness of functions. For instance, representing the fact that the size of a file thesis.tex is an unique value requires reasoning that we have LCW knowledge about size(thesis.tex) (a fact is placed in $D_C$) so that the planner can reason that it knows “all the values of size(thesis.tex)” The uniqueness of functions is important for the planner to be able to conclude, for example, that size(thesis.tex) $\neq$ 2048 when it knows size(thesis.tex) = 1024. The database descriptions presented in this thesis avoid this extra LCW information, by explicitly modelling functions as one of the basic types of knowledge an agent can represent and reason about.

Their approach to updating LCW information also results in LCW information being lost in circumstances where such a loss is not necessary. For instance, compressing a file thesis.tex causes the LCW information about the size of all the files in the directory with thesis.tex to be lost. Instead the approach taken in this thesis is to introduce a notion
of an *exception*, to mark the instances to which *LCW* information no longer applies. Further investigation of local closed world information has also lead to the separation and expansion of the *LCW* concept into distinct plan time and execution time constructs. Thus, the Internet Softbot work has influenced some of the notions presented in this thesis. This thesis, however, builds upon these ideas by extending or redefining them.

### 2.2 Knowledge and Planning

Moore’s work on planning provided a formal theory of the relationship between knowledge and action, and illustrated the importance of knowledge in an agent’s ability to plan to achieve a goal [Moo85]. Moore based his analysis on the realization that in the real world, planning and acting must frequently be done without complete knowledge. It is not sufficient for an agent to only consider what physical prerequisites are necessary for a plan, but also whether or not it has the information necessary to carry out the plan. If the agent lacks information then it must be able to reason about how to obtain the necessary information.

This challenged the classical assumption that if there is an action an agent is “physically” able to perform, and carrying out the action would result in achieving a goal, then the agent can in fact achieve the goal. Moore illustrated that this is not necessarily true by giving an example of a human agent attempting to open a safe. A human is physically capable of opening the safe, by the action of dialling the combination of the safe. But, it is not accurate to claim that an agent could open a safe simply by dialling the combination, unless he actually knew the combination. An agent should also possess the ability to reason about its knowledge and determine if a collection of facts implies knowledge of other facts.

Scherl and Levesque build on Moore’s work of knowledge and action [SL93]. The
primary focus of their work involved the development of a solution to the frame problem for knowledge-producing actions in the situation calculus. However, in addressing knowledge-producing actions they consider two sorts of actions: actions whose effect is to make known the truth value of some formula, and actions whose effect is to make known the value of some term. Their formalism develops an approach within the situation calculus that is able to model the changes a knowledge-producing action makes to the knowledge state of an agent.

These two types of sensory actions that are developed in their representation are also addressed in this thesis which investigates the importance of knowledge to planning, and the necessity of being able to model sensory actions. Unlike Scherl and Levesque, however, the formalism developed here is designed to support efficient reasoning and planning.

Levesque also points out some subtle issues concerning planning with incomplete information and sensing actions, that provide important insights into the problem [Lev96]. One of the important contributions Levesque makes is to propose a new definition of what a plan is in this setting, and when it is correct. He stresses the importance of a specification that is neutral and that does not use the terminology associated with particular planners or planning algorithms to describe the issues involved. Instead, the specification should be compatible with a wide range of planners and neutral to the choice of algorithm and data structure. He suggests formulating the specification of a planning task in terms of a general theory of action that includes sensing actions and the effect that they have on the knowledge of the agent executing them. The task of planning then becomes the task of not only finding the plan that achieves the goal, but also finding one that doesn’t depend on conditions whose truth values would be unknown to the agent at the required time.

This requirement places an important restriction on knowledge preconditions: that
at plan time the agent must know that the plan will achieve its desired effects, and at execution time the agent must have sufficient knowledge at every step of the plan to execute it. It must be known at plan time that the knowledge conditions in the plan will be achieved at execution time. This is essential for conditional planning problems where the action to perform next in a plan may depend on the result of an earlier sensing action (conditional planning is discussed further in Section 2.4).

Levesque’s work addresses some important issues about planning with incomplete knowledge that are explored in this thesis. The work presented here stresses modelling an agent’s knowledge, and emphasizes the separation of representation from planning algorithm. Some of Levesque’s ideas about plan correctness are also used. However, Levesque’s work does not directly address the issue of how plans can be generated. As well, Levesque’s model uses the situation calculus, requiring in general full first-order inference to reason about the effects of actions. This work focuses on a more limited representation that can be implemented more effectively in real planning systems.

2.3 STRIPS, ADL, and Database Representation

The work presented in this thesis is also motivated by STRIPS [FN71]. As mentioned in the introduction, a STRIPS-like representation is adopted, by using a collection of databases to represent an agent’s knowledge (as opposed to a single database as with STRIPS). Actions are also represented in a STRIPS-like manner by describing their parameters, preconditions, and the effects that they have on the databases, in terms of additions and deletions of information to the databases.

The representation in this thesis, however, goes further than the STRIPS representation by adopting some ADL style structures. ADL extends STRIPS by allowing operators to have conditional add and delete lists, functions to be updated, and arbitrary first-order
formulas to be used as preconditions [Ped89]. With ADL, conditional effects can be modelled. For instance, consider the action of dialling a combination on a safe. This action may have a conditional effect at plan time: an agent will come to know that the safe is open if it knows that the combination will open the safe. Similarly, this action may also have a conditional effect at execution time: the agent will know that the safe is open if the combination that it dialled actually opens the safe.

Reiter provides an approach related to planning, by formalizing the evolution of a database under update transactions [Rei92]. In database technology, the evolution of a database is determined by transactions, whose purpose is to update the database with new information. Transactions are procedures which physically modify data structures representing the current database state, much like the STRIPS operators do. Reiter points out that there is an important relationship between databases evolving under update transactions and dynamically changing worlds. In particular, he illustrates how proving properties of world states can be related to the integrity constraints placed on database states.

Even though Reiter’s application utilizes the situation calculus, this thesis addresses the same basic theoretical issue of trying to project an evolving set of databases to model changing worlds. The formal semantics for describing the items in each database provides the necessary translation into formulas of modal logic. Actions act as update operations on the databases, modifying their contents. Examples will illustrate that a wide range of actions can be modelled this way. Applying the effects of a sequence of actions to an initial collection of databases allows the initial knowledge state of an agent to be projected, modelling the agent’s representation of the world as it changes through action. Also, the representation is separated from the planning method, and is not restricted to a particular planning algorithm.
## 2.4 Contingency Planning

A complication arises when having to reason about how actions affect the agent’s knowledge. This is because the plan time effects of such actions are quite different from their execution time effects. For example, actions such as the UNIX command `ls` have different plan time and execution time effects that must be taken into consideration by the planner.

In many domains, a planner may only operate successfully if it is able to develop plans that take advantage of sensing information gathered during the execution of the plan. Levesque gives the example of an airport with two gates, and the goal of boarding the proper plane \cite{Lev96}. No sequence of actions can be shown to achieve the goal. The gate that the agent must go to depends on the execution time result of checking the departures board.

Such plans can be used to help manage uncertainty about the environment by allowing the planner to build plan sequences that are conditioned on the possible results of sensing actions executed earlier in the plan. During the execution of such a plan, the appropriate branch of the plan is executed if past sensing actions in the plan have provided information that makes the conditions of the branch true. These conditional or contingency plans give the planner the ability to plan ahead for a series of possible outcomes. The planner can prepare for contingencies by ensuring that the required items needed to execute more than one branch of the plan are collected beforehand and then use sensing information gathered at run time to guide the direction of the plan to the correct planned course of action for the given circumstance.

Decision points in conditional plans can be based on the information returned from a single sensing action (if `sense(P)` returns `P`, then perform branch `B1`, otherwise if `sense(P)` returns `¬P`, perform branch `B2`). In such plans, a sensing action causes a
branch point in the plan. In other plans, information from a number of sensing actions collected over time is used to make the decision of which branch should be executed (...sense(P)...sense(Q)...if P and Q then perform branch B1, else if P and \( \neg Q \) then perform branch B2, etc.).

Many classical planners such as STRIPS, however, develop unconditional plans that contain either no sensing actions, or else do not make use of the sensor information that is gathered from a plan, to direct the future course of that plan. Instead, they rely on the assumption that they have complete knowledge of the environment and of changes to the environment. However, uncertainty about the environment may preclude a planner from being able to generate a single course of action to accomplish a goal. To avoid replanning (or possibly simply failing) to achieve the goal, developing a set of plans for the projected contingencies that may arise may be preferred.

Pryor and Collins [PC96] point out that an effective contingency planner must have the capability to recognize when an uncertain outcome threatens the achievement of a goal, must be able to generate contingencies for all possible outcomes of the various sources of uncertainty that affect a given plan, schedule sensing actions that detect the occurrence of a particular contingency, and produce plans that can be executed correctly regardless of which contingency arises. Being able to enumerate contingencies while ensuring that its goal is achieved in every contingency is complex.

A number of planners have been developed that try to build conditional plans. CNLP is a conditional nonlinear planner developed by Peot and Smith [PS92]. CNLP uses STRIPS operators to describe the actions available to the planner. Besides the standard preconditions and effects specifications, actions may contain conditional effects. Such effects though must be explicitly enumerated in the operator description and are restricted to be a set of mutually exclusive outcomes.

The Internet Softbot’s planner, XII, also handles uncertainty in the environment by
2.4. CONTINGENCY PLANNING

using run-time variables to represent information as yet unknown in the plan [EHW⁺92, GEW96]. These variables are treated as constants without a known value, and allow the planner to build branches that arise from the information gathering steps of the plan. Branches make use of the run-time variables to construct plans that can achieve the goals in each case. XII also interleaves planning with execution. When a sensing action is executed the results that are gathered from the environment are then bound to the appropriate run-time variables, allowing the run-time variables in the plan to be “replaced” by these bound values. The actual process of interleaving the planning and execution however is not clear.

Draper, Hanks, and Weld [DHW93] address the problem of managing incomplete information and knowledge-producing actions by using probabilistic planning in the development of the C-BURIDAN planner. C-BURIDAN is able to represent sensory actions (including noisy sensors), and has a plan representation and generation algorithm that supports contingency planning. However, their approach is probabilistic while the approach presented in the thesis is not.

Cassandra, a partial-order contingency planner developed by Pryor and Collins uses specific decision steps to determine which of the possible courses of action it should pursue [PC96]. Information gathering steps are distinct from decision steps, thus the process of gathering information is distinguished from the process of making decisions. Circumstances in which it is possible to perform an action are distinguished from those in which it is necessary to perform it. This allows decision points to depend on information from a collection of actions, as opposed to the value of a single sensing action. Uncertainty is introduced into a plan when the plan becomes dependent on the outcome of an action that has an uncertain effect. At this point, contingencies are developed for each possible outcome of the uncertainty.

Pryor and Collins also point out some important issues concerning contingency plan-
A plan with no contingencies may not always be superior to a plan that has contingencies. For example, consider a programming agent that has the goal of locating a library that is needed to compile a program. A contingency plan to achieve the goal could first check the local file system to see if the library is available. If so, the local copy can be used. Otherwise the source code for the library can be downloaded from the Internet and the library can be built. A plan with no contingencies may only have the actions of downloading the source code for the library and building it. Clearly such a plan is less desirable than the contingent plan, if it needs to be executed each time the library is needed. It is also possible that a single plan could work just as well for several different outcomes of an uncertainty. However, a planner building contingency plans may be inefficient in that it may have to “find” the same plan multiple times.

Contingency planning does present some problems. The planner may not be able to achieve the goal along all the branches of the conditional plan. In this case, a planner may still fail, even though it is able to develop plans that achieve the goal in some of the contingencies, though not all of them. A greater problem is that in many domains, conditional plans end up being too large [PS92, PC96]. Plan steps can possibly expand exponentially given the number of decision points and number of contingencies to plan for at each decision point. Branch merging is a possibility, to construct a plan that splits and then reunites, but this adds further complications to the planning process.

An agent often has to commit to part of a plan by actually executing some actions to rule out contingencies and avoid having to plan for them. While interleaving planning and execution does have the advantage of not having to plan for contingencies that do not arise, there are also problems. Without being able to plan further in advance, the planner may find suboptimal plans. A more dangerous possibility, however, is that actions, (even some sensing actions) may change the world, with the result that it is no longer possible to achieve the goal. In some domains, replanning may be possible to restore the condi-
2.4. CONTINGENCY PLANNING

tions of the world [GEW96], allowing planning to continue, but in general, this problem remains an important concern.

The issue of planning is not directly addressed in this thesis. Instead, an underlying representational formalism that supports planning is developed. In particular, this representation supports the projection of action sequences. That is, we can reason about the effects of an action sequence on the world and on the agent’s knowledge. Once projection is achievable, planning becomes possible by searching for an appropriate action sequence. Contingency planning can also be supported. Furthermore, the representational formalism that is developed here makes a careful distinction between plan time and execution time effects. Thus, it can support planning through the projection of the plan time effects, and it can also support plan execution through the projection of the execution time effects. In fact, one of the important features of this representation is that it allows for further investigation of the tradeoffs between planning and execution. Interleaving planning and execution remains an important open problem in the area, and this work tries to open up a wider range of possibilities for addressing this problem.
Chapter 3

Representing the Agent’s Knowledge

The first issue that is addressed in this thesis is that of representing the agent’s knowledge. In particular, the focus is on the situation where an agent has correct, albeit incomplete, information about the environment. In considering this problem, a number of questions must be addressed. For instance, decisions must be made as to the types of knowledge to represent, and what restrictions are placed on the representation. As well, representing knowledge-producing actions is a concern. It is necessary to develop a representation that can model sensing actions as well as the knowledge that an agent will come to know at some point in the future. Moreover, the representation should be formally characterized, but without restricting its use in actual planning systems. A representational formalism is developed in this chapter that attempts to meet these concerns.

3.1 Extending the STRIPS Representation

Developing an effective representation is essential to the development of an effective planning system. As a result, consideration must be made in deciding what kinds of knowledge can be represented, and how it should be represented. For instance, at this
time, we do not know how to deal with a fully general logic of knowledge.

Recall from Chapter 1 that the STRIPS representation made use of a single database, $D$, to represent an agent’s knowledge. However, the standard STRIPS representation also uses the closed world assumption and is very restrictive in the types of knowledge that can be represented. STRIPS does offer a straightforward representation scheme though, and an efficient method of updating the representation and reasoning from it. Chapter 1 also illustrated how an agent’s knowledge could be characterized by using a standard modal logic of knowledge, and that the contents of the STRIPS database could be formally described by modal logic formulas.\(^1\)

A STRIPS like approach is adopted in this thesis for representing an agent’s incomplete knowledge. Instead of using a single database, the representation is extended and a collection of databases is used. Each database will be used to maintain a particular type of knowledge. As with the STRIPS database, the agent’s knowledge will be formally characterized by providing a translation from the database contents to a set of logical formulas in modal logic. Thus, the logic’s semantics can be used as the underlying semantics of the representation. In essence though, we are restricting ourselves to a particular subset of the logic, the subset that can be represented as the contents of the databases.

When referring to the representation, $DB$ will be used to represent the collection of databases, and $KB$ will be used to represent the set of logical formulas that characterize the agent’s knowledge.

---

\(^1\)See [FHMV95] for a good introduction to modal logic and reasoning about knowledge.
3.2 Knowledge, Rigid Terms, and Constant Domains

Chapter 1 provided an introduction to modal logic. For example, a formula such as $K(\phi)$ can be used to represent the notion that “the agent knows $\phi$”. One of the requirements of our representation though is that the agent’s knowledge of the world is correct. Thus, if an agent knows some information about the world, that information should in fact be true in the real world. In terms of modal knowledge, the assumption of correct knowledge is modelled by the fact that the real world, $w^*$, is a member of the set of worlds $W$ the agent considers possible. Since every world is accessible from every other world, if the agent’s knows $\phi$, then $\phi$ is true at all the worlds in $W$, and so $\phi$ must be true at $w^*$.

As a convention, all logical formulas will be interpreted at $w^*$. Thus a formula such as $K(\text{readable}(\text{thesis.tex})) \land \text{writable}(\text{thesis.tex})$ means that the agent knows that the file thesis.tex is readable and by the semantics of $K$, that thesis.tex is actually readable in the real world. It is also true in the real world that the file is writable, though this is not necessarily known by the agent.

The agent’s knowledge will include atomic facts about various terms. For instance, an agent may know that the file thesis.tex is readable. This knowledge might be represented by the atomic formula $K(\text{readable}(\text{thesis.tex}))$, where thesis.tex is a term of the language. Functions are also allowed in the agent’s knowledge, providing a mapping between terms. For example, the agent might know various function values like $K(\text{size}(\text{thesis.tex}) = 8192)$, representing the fact that thesis.tex is 8192 bytes in length.

When dealing with knowledge, terms that are composed from functions and constants, such as thesis.tex, size(thesis.tex), and 8192, pose potential problems. In particular, the terms that they generate may be non-rigid or rigid. Non-Rigid terms are terms whose denotation varies from world to world. For instance, an agent might not know the size of the file thesis.tex, so the term size(thesis.tex) may have a different denotation.
(i.e., a different value) in every world the agent considers possible. On the other hand, rigid terms have a fixed denotation across worlds. Thus, a number like 8192 would have the same denotation (i.e., the same meaning) in every world.

Reasoning about facts like readable(thesis.tex) becomes more complex when terms can be either rigid or non-rigid.\textsuperscript{2} For instance, if the term thesis.tex had potentially different denotations in every world, it isn’t immediately obvious what this fact would mean to an agent. To avoid this problem and simplify the representation, all constants are restricted to being rigid. Thus, a term like thesis.tex will always denote the same object in every world the agent considers possible.\textsuperscript{3} Furthermore, an unique names assumption is utilized. If \( c_1 \) and \( c_2 \) are syntactically distinct constants, then they must denote different objects. In other words, \( c_1 \neq c_2 \). The representation is not greatly restricted by this assumption, since there does not seem to be a need for the level of generality provided by non-rigid constants. Functions, however, are allowed to generate non-rigid terms. Thus, a term like size(thesis.tex) can denote a different value in different worlds. The assumption is made though that numeric functions, like “+”, and numeric predicates, like “>”, are rigid and have their standard interpretation in every world.

Formally, this means that for every constant \( c \) in the language describing any particular planning domain, the agent’s knowledge (the set \( \text{KB} \)) includes the formula:

\[
\exists x. K(x = c).
\]

\textsuperscript{2}See Garson [Gar84] for a good discussion of these issues.

\textsuperscript{3}Many files in the agent’s environment might be called thesis.tex. In practice, each file would have to be identified by a distinct constant. One way to do this is to use an unique identifier for each file and have a function that maps this identifier to the “common” name of the file. The function would still allow files with different identifiers to be mapped to the same common name. For readability, common names will still be used in the examples; however, the reader should realize that these names are intended to represent unique objects.
This says that there is a particular object in the real world such that in every possible world the constant $c$ denotes that object. In addition, for every two syntactically distinct constants $c_1$ and $c_2$, $\text{KB}$ includes the formula:

$$K(c_1 \neq c_2). \quad (3.2)$$

This says that in all worlds, syntactically distinct constants denote different objects.

Another possible complication concerns a world’s domain of discourse. In a \textit{fixed domain} interpretation, every world has the same domain of discourse (i.e., the same set of objects). On the other hand, in a \textit{world-relative} interpretation different worlds may have different domains. Thus, in a world-relative domain there may be worlds in which \texttt{thesis.tex} is a member of that world’s domain, and there may also be worlds in which the object \texttt{thesis.tex} does not exist. Reasoning becomes more difficult under such circumstances.\(^5\) To simplify reasoning, the semantics are restricted to only consider models in which all the worlds have an identical domain of discourse.\(^6\)

The assumption of identical domains of discourse makes the Barcan formula valid \[\text{Gar84}\]:

$$\forall \bar{x}.K(\phi(\bar{x})) \Rightarrow K(\forall \bar{x}.\phi(\bar{x})).$$

Suppose that there are only two objects in the domain of discourse, $x_1$ and $x_2$, and that the agent knows $\phi(x_1)$ and $\phi(x_2)$. Thus, it is true that $\forall x_i. K(\phi(x_i))$. If the domains are identical across all worlds, then by the Barcan formula it also follows that $K(\forall x_i.\phi(x_i))$

\(^4\)Actually, the rigidity of constants ($\exists x. K(x = c)$) along with their inequality in the real world ($c_1 \neq c_2$) is sufficient to derive their inequality in all worlds ($K(c_1 \neq c_2)$). Equation 3.2 is included in $\text{KB}$ for convenience.

\(^5\)Again, see \[\text{Gar84}\] for a discussion of the issues concerning fixed versus world-relative domain models.

\(^6\)This assumption has not created any practical problems. In particular, it does not mean that we necessarily know the identity of all the objects in the real world.
is true. That is, the agent knows that in all worlds for every object $x_i$, the formula $\phi(x_i)$ holds. However, without the assumption of identical domains, this may not be true. For instance, there may exist worlds whose domains contain a third object, $x_3$, such that $\phi(x_3)$ does not hold. Thus, the consequent of the Barcan formula, $K(\forall x_i.\phi(x_i))$, would not be valid in these cases.

3.3 The Databases

An agent’s knowledge is represented by a collection of four databases, each of which will be discussed below. A description of the types of knowledge that can be represented, along with any restrictions placed on the knowledge, will be provided. Moreover, a formal characterization of the database contents will be stated.

3.3.1 The $K_f$ database

The first database, $K_f$, is much like a standard STRIPS database, except that the closed world assumption is not applied and so both positive and negative facts may be explicitly represented. $K_f$ can include any ground literal (an atomic formula or the negation of an atomic formula) of the form $l(c_1, \ldots, c_n)$, where all the $c_i$ are terms that are constants. Thus, facts like $\text{readable}(\text{thesis}.\text{tex})$ (“the file thesis.tex is readable”) and $\neg\text{writable}(\text{source}.\text{c})$ (“the file source.c is not writable) are valid entries in $K_f$. However, an atomic formula like $\text{readable}(\ldots(\text{project}))$, where the function “..” specifies the parent directory of a directory named project, cannot appear in $K_f$. To include such information, the name of project’s parent directory would have to be known.

In addition to ground literals, $K_f$ can also contain equality literals specifying function values. In particular, $K_f$ can contain formulas of the form $f(c_1, \ldots, c_n) = c_{n+1}$, where $f$ is an $n$-ary function and the $c_i$ are all constants. This formula specifies that $f$’s value on
3.3. THE DATABASES

this particular set of arguments is the constant $c_{n+1}$. This restriction means that function values in $K_f$ are considered to be known by the agent only if they can be “grounded” out as constant values. For example, if the agent knew that the parent directory of the directory project was the directory coursework, then a function value such as ..(project) = coursework could be contained in $K_f$.

The contents of $K_f$ can be specified in terms of the agent’s knowledge. Formally, for every formula $\ell \in K_f$, $KB$ (the set of logical formulas characterizing the agent’s knowledge) includes the formula:

$$K(\ell).$$

(3.3)

3.3.2 The $K_w$ database

The second database, $K_w$, contains a collection of formulas, every instance of which the agent either knows or knows the negation of (but may not know which). In particular, $K_w$ can contain any formula that is a conjunction of atomic formulas, including formulas that contain free variables. Simple ground atomic facts added to $K_w$ can be used to model the plan time effects of sensing actions. For instance, consider an agent that is reasoning about executing a sensing action that determines if the file thesis.tex is readable or not. Adding the fact \texttt{readable(thesis.tex)} to $K_w$ can be used to model the plan time effect of this sensing action: at plan time all the agent knows is that after sensing, it will know whether or not this fact is true. However, the agent is unable to resolve this disjunction until execution time.

By adding formulas containing variables to $K_w$, actions that generate universal plan-time effects can be modelled. For instance, an action whose effects yield local closed world information [EGW97] can be represented using this database. Consider the UNIX command \texttt{ls dir}, that provides local closed world information about the names of the
files in the directory \textit{dir}. An agent reasoning about such a command at plan time will not know the actual contents of the directory, simply that this information will become known after the \textit{ls} action is executed. Adding a formula such as \textit{in-dir}(x, \textit{dir}) to $K_w$ can represent the plan time effects of the \textit{ls} action (where \textit{in-dir}(x, y) indicates that file \textit{x} is in directory \textit{y}). This formula indicates that the agent knows every file \textit{x} that is in directory \textit{dir}.

Some special predicates, for instance numeric predicates like $<$ and equality $=$, are rigid, having the same denotation in every world in $W$. These predicates are considered to be implicit members of $K_w$. For example, the formulas $x < y$ and $x = y$ are implicitly in $K_w$.\footnote{The inference algorithm presented in the next chapter has access to these implicit members of $K_w$.}

The contents of $K_w$ can be specified formally in terms of the agent’s knowledge. For every formula $\phi(\vec{x}) \in K_w$ (a conjunction of atomic formulas in which the variables in $\vec{x}$ appear free), $KB$ includes the formula:

$$\forall \vec{x}. K(\phi(\vec{x})) \lor K(\neg\phi(\vec{x})).$$

A useful notation is the formula $K_{\text{whe}}(\phi)$ which is defined to be the formula $K(\phi) \lor K(\neg\phi)$: either $\phi$ or its negation is known to hold.

### 3.3.3 The $K_v$ database

The third database, $K_v$, is a specialized version of $K_w$ designed to store information about the various function values the agent will come to know. Formulas in $K_v$ may be of the form $f(t_1, \ldots, t_n)$, where $f$ is a function whose arguments, $t_i$, are non-function terms. For example, $f(x, c)$ would be a legal entry in $K_v$ while a nested function term such as $f(g(b), a)$ would not be.
3.3. **THE DATABASES**

Entries in $K_v$ can also be used to model sensing actions. Whereas $K_w$ could be used to model sensors that return truth values, the sensors in this case return constants (for instance, numbers). A function entered into $K_v$ can represent the fact that the value returned by the sensor will be known at execution time, but at plan time the agent only knows that this value will become known. For instance, placing a function such as $\text{size}(\text{thesis.tex})$ into $K_v$ can model a plan time effect of an action, such as the UNIX command $\text{ls -al thesis.tex}$, that determines the length of the file `thesis.tex` in bytes.

The contents of $K_v$ can be represented formally in terms of the agent’s knowledge by specifying that for every formula $f(\bar{x}) \in K_v$, where $\bar{x}$ is the set of variables appearing in the term, $\mathbf{KB}$ includes the formula:

$$
\forall \bar{x}. \exists v. K(f(\bar{x}) = v).
$$

(3.5)

Formulas of this type are a standard way of specifying that the agent knows a function value.\(^8\)

More general information about knowing function values can be specified by entries in $K_w$. For example, consider the UNIX command $\text{ls -al dir}$. This action has the effect that the agent will come to know the sizes of all the files in a particular directory $\text{dir}$. To represent this information, the conjunctive formula $\text{in-dir}(x, \text{dir}) \land \text{size}(x) = y$ can be placed in $K_w$. This formula says that for every file $x$ that is in directory $\text{dir}$, the agent knows all the values of $y$ such that $\text{size}(x) = y$. Since $\text{size}$ is a function, there is only one such $y$ for each $x$.

\(^8\)See Scherl and Levesque [SL93].
3.3.4 The LCW database

The fourth database, LCW, is a database of local closed world information. Formulas in LCW have a similar form to formulas in $K_w$: LCW can contain any formula that is a conjunction of atomic formulas, including formulas that contain free variables. A conjunctive formula $\phi$, in LCW, essentially asserts that the agent’s $K_f$ database contains a complete list of all items that satisfy $\phi$. LCW can be thought of as the execution time analog of $K_w$. Whereas formulas in $K_w$ are used to represent plan-time effects, LCW formulas can represent the universal effects an action generates during execution. For instance, consider again the UNIX command `ls dir`, that returns a list of all the files in the directory `dir` (cf. the $K_w$ database, Section 3.3.2). Executing this action has the effect of generating a list of facts, such as `in-dir(file1, dir)`, `in-dir(file2, dir)`, etc., that are added to $K_f$. To capture the closed world information that this list is complete (the notion that all the files in directory `dir` are listed in $K_f$), a formula such as `in-dir(x, dir)` is added to LCW. Since an LCW formula infers that the listing of items that satisfy it is complete, this representation can be used to determine the truth of any ground instance of the formula.

The formal semantics of the LCW database are specified as follows. Let $\phi(\bar{x}) = \alpha_1(\bar{x}) \land \ldots \land \alpha_k(\bar{x})$ be a conjunction of atomic formulas in which the vector of variables $\bar{x} = (x_1, \ldots, x_n)$ appear free. Say that $\phi(\bar{x}) \in LCW$. Let $C = \{ \bar{c} : \alpha_i(\bar{x}/\bar{c}) \in K_f, 1 \leq i \leq k \}$. $C$ is the set of tuples of constants explicitly listed in $K_f$ as satisfying $\phi$. For

---

9In most cases, a list of facts such as the list of filenames given in the example, can only be added to the $K_f$ database when an action is actually executed, hence the notion of LCW as an execution time representation.

10Note that not every variable in $\bar{x}$ need appear free in every literal $\alpha_i$. 

3.4. **THE RELATIONSHIP BETWEEN \( K_w \) AND LCW**

every such formula \( \phi \in LCW \), \( KB \) includes the formula:

\[
\forall \vec{x}. \bigwedge_{\vec{c} \in C} \neg(x_1 = c_1 \land \ldots \land x_n = c_n) \Rightarrow K(\neg\phi(\vec{x})).
\]

Consider the example where \( P(x, y) \land Q(x) \in LCW \), and \( P(a, c), P(a, d), Q(a) \) and \( Q(b) \) are all in \( K_f \). By the formal semantics of \( LCW \), the set \( C \) contains the pairs \((a, c)\) and \((a, d)\), since these two pairs of constants are explicitly listed in \( K_f \) as satisfying \( P(x, y) \land Q(x) \). Thus, the formula:

\[
\forall x, y. \neg(x = a \land y = c) \land \neg(x = a \land y = d) \Rightarrow K(\neg(P(x, y) \land Q(x)))
\]

is in \( KB \). This formula represents the closed world information that the pairs \((a, c)\) and \((a, d)\) are in fact the only pairs satisfying \( P(x, y) \land Q(x) \). As a result, the formula also entails a collection of negative instances, for example that \( K(\neg(P(d, c) \land Q(d))) \), \( K(\neg(P(c, a) \land Q(c))) \), etc.

The formalization of \( LCW \) presented here makes explicit the notion utilized by Etzioni et al. [EGW97], that if an agent has local closed world information about a formula, and an instance of that formula is not explicitly listed in the \( K_f \) database, then the agent can conclude that the formula does not hold for that particular instance.

### 3.4 The Relationship Between \( K_w \) and \( LCW \)

The \( K_w \) and \( LCW \) databases both manage the local closed world information of an agent. As a result there is a close relationship between the two databases. A semantics for the \( LCW \) and \( K_w \) databases was provided by translating their contents to modal logic formulas. Recall from the discussion of the \( LCW \) database, that if a formula \( \phi(\vec{x}) \) is in \( LCW \), then the agent knows a list of satisfying instances of the formula (the set \( C \)) in \( K_f \).
When entries in $K_w$ are converted to formulas of the form $\forall \bar{x}. K(\phi(\bar{x})) \lor K(\neg\phi(\bar{x}))$, this corresponds to the agent knowing that the set of satisfying instances of $\phi(\bar{x})$ is invariant across worlds. That is, a tuple of constants $\bar{c}$ satisfies $\phi(\bar{x}/\bar{c})$ in the real world if and only if it satisfies the formula in every world the agent considers possible.

Unlike the LCW database, the presence of such a formula in $K_w$ does not mean that the agent knows the truth value of $\phi(\bar{x}/\bar{c})$. It is not until the action is actually executed that this formula is resolved. After execution, the set of satisfying instances will have been added to $K_f$. The formula $\phi(\bar{x})$ entered in LCW can represent that this list is complete. Hence, the agent will know the truth value of $\phi(\bar{x}/\bar{c})$ for every $\bar{c}$. Thus the specification for an action that has a local closed world effect will typically include a plan time addition to the $K_w$ database, and an execution time addition to LCW.

The approach of Etzioni et al. to managing LCW information is primarily an execution time representation. For instance, the inference algorithm that they have developed in [EGW97] is an execution time algorithm that requires the actions executed to add their satisfying instances to the database of facts. An agent is unable to perform LCW reasoning at plan time with this approach as these satisfying instances are not yet known. Splitting the local closed world information into plan time effects (modelled by formulas in $K_w$) and execution time effects (modelled by formulas in LCW) gives the agent the ability to perform a form of LCW reasoning at plan time, as opposed to simply at execution time.

### 3.5 The Knowledge State

Now that the contents of the databases have been formally described, it is possible to define the knowledge state of an agent. Recall from Chapter 1, that the knowledge state of an agent using the STRIPS database, $D$, was defined as the conjunction of the logical
formulas that were translated from the contents of $\mathcal{D}$. This approach is also used to handle the collection of four databases. Given a particular set of these four databases, that is, a particular $DB$, the agent’s knowledge state is defined by the set of formulas in $KB$, as specified by the Formulas 3.1–3.6, that formally describe the contents of the databases. In particular, the agent’s knowledge state is characterized by the set of models (in which every possible world has the same domain of discourse) that satisfy all of the formulas in $KB$.

Now that the agent’s knowledge state has been defined, a logical question to ask is: under what conditions is this knowledge state consistent? Since the knowledge state of the agent is directly dependent on the contents of the databases, an inconsistent knowledge state can be determined by specifying the conditions under which a collection of databases (a $DB$) give rise to an inconsistent $KB$. The following theorem answers this question and shows that subject to two obvious consistency requirements, any $DB$ specifies a consistent $KB$.

**Theorem 3.5.1** Let $DB$ be any set of the four databases subject to the two conditions that:

1. there is no atomic formula $\alpha$ with both $\alpha$ and $\neg \alpha$ in $K_f$, and
2. no function $f(c_1, \ldots, c_n)$ is specified to have two distinct values in $K_f$.

Then the $KB$ corresponding to $DB$ is consistent. That is, $KB$ has a model.

**Proof:** In general $KB$ will have many models. We show how a particular model can be constructed. First, we let the domain of discourse be the set of all constants appearing in $DB$. Then start with the set of ground literals (and function values) contained in $K_f$. Then we add to $K_f$ a set of negative facts sufficient to satisfy all of the formulas arising
CHAPTER 3. REPRESENTING THE AGENT’S KNOWLEDGE

from \( LCW \). Let

\[
\forall \vec{x}. \bigwedge_{\vec{c} \in C} \neg(x_1 = c_1 \land \ldots \land x_n = c_n) \Rightarrow K(\neg\phi(\vec{x}))
\]

be a formula in \( KB \) arising from a formula \( \phi \in LCW \), where \( C \) is the list of tuples \( \vec{c} \) such that \( \phi(\vec{x}/\vec{c}) \in K_f \). For every \( \vec{c} \notin C \) we pick a conjunct of \( \phi(\vec{x}/\vec{c}) \), \( \alpha_i(\vec{x}/\vec{c}) \), that is not in \( K_f \): one such conjunct must exist by the definition of \( C \). In fact, more than one such conjunct may exist, in which case we make an arbitrary choice. We add \( \neg\alpha_i(\vec{x}/\vec{c}) \) to \( K_f \), thus satisfying that negative instance of \( \phi \). We do this for every negative instance of every \( \phi \in LCW \).

Note that since no positive facts are added to \( K_f \) by this process, our additions do not affect what we can infer from \( LCW \) using this augmented \( K_f \) (the sets \( C \) of satisfying instances do not change). Hence, the addition of negative facts to \( K_f \) in order to satisfy a formula \( \phi \in LCW \) do not affect the additions required to satisfy any other formula \( \phi' \in LCW \).

Clearly, the resulting set of facts in \( K_f \) continues to satisfy the two conditions of the theorem, and thus this set of facts has at least one first-order model. We pick an arbitrary model, \( w \). Finally, we build a model for the modal logic by setting the collection of possible worlds \( W \) to be simply the set \( \{w\} \). It is not difficult to see that this set of worlds \( W \) satisfies any formula of the form \( \forall \vec{x}.K(\phi(\vec{x})) \lor K(\neg\phi(\vec{x})) \) that could arise from entries in \( K_w \) and \( K_v \).

Knowing the conditions under which a \( KB \) is consistent is important in designing an agent that can act in the world. Actions update the agent’s representation of the world, and so, alter the agent’s knowledge state. If an action specification is to be developed, it must ensure that the agent’s knowledge remains consistent when the action updates its knowledge state. Otherwise, an agent may end up with an inconsistent knowledge state.
where it can infer anything.\textsuperscript{11} The consistency requirements from Theorem 3.5.1 lead to the following corollary:

**Corollary 3.5.2** *If actions are specified as additions and deletions to the four databases, and these updates maintain the consistency conditions stated in Theorem 3.5.1, then no sequence of actions can give rise to an inconsistent KB.*

The result is that the formalism presented here is very similar to the classical STRIPS representation. For instance, in STRIPS, any database is logically consistent and any sequence of actions (also specified as additions and deletions to the database) maintain consistency. Like the STRIPS representation, the automatic maintenance of consistency has both positive and negative features. On the positive side, a user of the representation never needs to worry about “breaking” the representation by generating an inconsistent state. On the negative side, the onus is on the user to build an accurate domain model. This is similar to what is required when using STRIPS. The user must ensure that that the KB represented by the databases makes sense in the domain being modelled and that actions update KB in a sensible manner. For example, an agent in the Blocksworld domain can’t be carrying an object and have its hands empty at the same time. The user must ensure that the databases representing the initial world satisfies these state constraints, and that the actions properly update the databases so as to maintain those constraints.

\textsuperscript{11}Everything is entailed by logical inconsistency.
Chapter 4

Inference

The previous chapter described the types of knowledge an agent could represent, along with a formal semantics for defining the knowledge state of the agent. In this chapter an inference procedure is presented for querying the agent’s knowledge. An inference procedure provides a means for testing if the agent’s knowledge satisfies certain knowledge conditions, such as those required to select the appropriate actions during the planning process. Proofs are given that show that this algorithm is sound (the inferences it makes are correct), but incomplete (it cannot infer every logical consequence of the agent’s knowledge). Finally, a simple query language is defined, based on a set of primitive queries that make use of the inference procedure.

4.1 Sound, Complete, and Incomplete Inference

An agent’s knowledge is represented by the collection of databases, DB. If an agent is to effectively use its knowledge to improve its performance in the world, it needs to reason about what conclusions can be inferred from DB. Inference, though, not only involves verifying that an agent knows (or doesn’t know) some fact, by looking it up in the proper
database, but also requires a means of determining whether or not some formula logically follows from the set of formulas in DB.

An inference procedure describes in an algorithmic way, the process by which an agent can infer conclusions from its collection of databases. The inference procedure that is described in this chapter operates by answering queries about the agent’s knowledge: given a formula $\varepsilon$, it tries to determine from its set of rules what it can infer about $\varepsilon$ from DB.

An agent’s knowledge state, $\text{KB}$, is based on its collection of databases, $\text{DB}$. As a result, there is a correspondence between conclusions that are inferred from $\text{DB}$, and conclusions that are logically entailed by $\text{KB}$. An inference procedure is said to be sound if whenever the procedure infers a formula $\varepsilon$ from $\text{DB}$, then $\text{KB} \models \varepsilon$. Soundness is a correctness condition: any inference the procedure makes is supported logically by the agent’s knowledge state. An inference procedure is complete if whenever $\text{KB} \models \varepsilon$, then $\varepsilon$ can be inferred from $\text{DB}$ by the inference procedure. If there exists a formula $\varepsilon$ such that $\text{KB} \models \varepsilon$ but $\varepsilon$ cannot be inferred from $\text{DB}$ by the inference procedure, then the inference procedure is said to be incomplete.

Inference procedures that are complete are not always desirable. Such procedures require the capability to infer all the logical consequences of the agent’s knowledge. Among these logical consequences are all tautologies. Requiring an agent to infer all logical truths is the problem of logical omniscience [Hin75]. There is no tractable procedure for making such inferences in a first-order language like the one used here.\footnote{In a propositional logic, deciding if a formula follows from a propositional $\text{KB}$ is co-NP (it is the dual of the unsatisfiability problem). In a first-order logic the problem is only semi-decidable: there exists a procedure that will eventually return True if the formula is entailed by $\text{KB}$, but it may never return if the formula is not entailed.}

Complete inference can often be avoided, especially in a planning setting. For instance, planners primarily require the ability to decide whether or not some atomic for-
mula is true or false at a given point in the plan. A planner that reasons with incomplete knowledge may also need to determine at some point in the plan, whether or not it will $K_{\text{wh}}$ some fact. These kinds of inferences can often be made without requiring the inference procedure to be complete.

### 4.2 Inference Algorithm

The inference procedure that is presented here is able to answer queries about ground atomic formulas or variable free terms. The inference procedure consists of two parts, a term evaluation algorithm and the actual inference engine. The term evaluation algorithm, EvalTerm, tries to simplify the input before it is passed to the inference algorithm. It attempts to evaluate functions that appear in the input, and that are also known to the agent, by replacing the function with the constant value it is known to be equivalent to.

The actual inference is done in the algorithm $IA$, by following a set of rules that can access the contents of the agent’s databases.\(^2\) The inference algorithm returns one of four outputs about a query $\varepsilon$: T, F, W, or U, that represent the conclusion it was able to make about the input. If the inference algorithm returns T, this indicates that it was able to determine that $KB \models K(\varepsilon)$. That is, the agent can infer that it knows $\varepsilon$. An output of F indicates that $KB \models K(\neg \varepsilon)$: the agent knows the negation of $\varepsilon$. When $IA$ returns W, this implies that $KB \models K_{\text{wh}}(\varepsilon)$ when $\varepsilon$ is a formula, and that $KB \models \exists x. K(x = \varepsilon)$ when $\varepsilon$ is a term. When $\varepsilon$ is a formula, this means that the agent has “know whether” knowledge of $\varepsilon$. When $\varepsilon$ is a term, this indicates that the agent knows, or will come to know, the value of the term. An output of U indicates that $IA$ is unable to conclude anything about $\varepsilon$. If the input to $IA$ is a single term, then the algorithm only responds with an output of W or U. The inference algorithm is presented in Table 4.1.

\(^2\)The inference algorithm also has access to the implicit members of $K_{\text{w}}$, such as numerical predicates.
CHAPTER 4. INFERENCE

Procedure $\text{IA}(\varepsilon)$

**Inputs:** Either a ground atomic formula containing the terms $t_1, \ldots, t_k$, or a single term. The terms in $\varepsilon$ can contain functions but no variables.

**Output:** $T$, $F$, $W$, or $U$ subject to the conditions: (1) $T$ implies $KB \models K(\varepsilon)$, (2) $F$ implies $KB \models K(\neg \varepsilon)$, (3) $W$ implies $KB \models \exists x.K(x = \varepsilon)$ when $\varepsilon$ is a term, and (4) $U$ implies the algorithm is unable to conclude anything about $\varepsilon$.

1. Replace each $t_i$ in $\varepsilon$ by $\text{EvalTerm}(t_i)$. Call this simplified formula $\varepsilon'$.

2. If $\varepsilon'$ is the term $t$ and either (1) $t$ is a constant or (2) there exists a $t_0 \in K_v$ and a substitution $\theta$ such that $t\theta = t$, then return $(W)$. Else return $(U)$.

3. If $\varepsilon'$ is of the form $t_1 = t_2$, then if these two terms are syntactically identical return $(T)$. Else if $t_1$ and $t_2$ are both constants then return $(F)$. Else return $(U)$.

4. If $\varepsilon' \in K_f$, then return $(T)$.

5. If $\neg \varepsilon' \in K_f$, then return $(F)$.

6. If there exists a $\phi(x) = \alpha_1(x) \land \ldots \land \alpha_k(x) \in LCW$ and a ground instance of $\phi$, $\phi(x/\vec{a})$, such that (1) $\vec{a}$ are constants appearing in $K_f$, (2) $\alpha_i(x/\vec{a}) = \varepsilon'$ for some $i$, and (3) $\text{IA}(\alpha_j(x/\vec{a})) = T$ for all $j \neq i$, then return $(F)$.

7. If there exists a $\phi(x) = \alpha_1(x) \land \ldots \land \alpha_k(x) \in K_w$ and a ground instance of $\phi$, $\phi(x/\vec{a})$, such that (1) $\vec{a}$ are either constants appearing in $K_f$ or terms $t_i$ appearing in $\varepsilon'$ for which $\text{IA}(t_i) = W$, (2) $\alpha_i(x/\vec{a}) = \varepsilon'$ for some $i$, and (3) $\text{IA}(\alpha_j(x/\vec{a})) = T$ for all $j \neq i$, then return $(W)$.

8. Else return $U$.

Procedure $\text{EvalTerm}(t)$

**Inputs:** A variable free term.

**Output:** $t'$ the simplest term known to be equal to $t$.

1. If $t$ is a constant then return $(t)$.

2. If $t = f(t_1, \ldots, t_k)$ and $f(\text{EvalTerm}(t_1), \ldots, \text{EvalTerm}(t_k)) = c \in K_f$ or we can compute that $f$ on these arguments is equal to $c$ (e.g., when $f$ is an arithmetic function) then return $(c)$, else return $(f(\text{EvalTerm}(t_1), \ldots, \text{EvalTerm}(t_k)))$.

Table 4.1: Inference algorithm
Consider a simple example of the operation of the inference procedure $\text{IA}$ on the query $\text{IA}(\text{in-dir(thesis.tex, home-dir(papers)))}$, when the agent’s $K_f$ database contains the formulas $\text{in-dir(thesis.tex, /project)}$ and $\text{home-dir(papers)} = /\text{project}$, and the other databases are empty. In this case, $\text{IA}$ will return $\text{T}$. In the first step, $\text{EvalTerm}$ tries to simplify the input term, $\text{home-dir(papers)}$. Since there is a function, $\text{home-dir(papers)} = /\text{project}$ in $K_f$, the term $\text{home-dir(papers)}$ is replaced with $/\text{project}$. The query reduces to $\text{in-dir(thesis.tex, /project)}$. The algorithm proceeds until step 4, where the $K_f$ database is checked. Since $\text{in-dir(thesis.tex, /project)}$ is in $K_f$, the algorithm returns $\text{T}$.

A more complex example of the operation of $\text{IA}$ is the query $\text{IA}(\text{size(thesis.tex)} > 30000)$ when $\text{size(thesis.tex)} \in K_v$ is the only entry in the databases. In this situation, $\text{IA}$ will return $\text{W}$. First, $\text{EvalTerm}$ tries to simplify the input term $\text{size(thesis.tex)}$. In this case no simplification can be made since this function is not in $K_f$. Because there are no formulas in $K_f$ or $LCW$, $\text{IA}$ continues to step 7. From Section 3.3.2, the predicate $>$ is rigid and thus the formula $x > y$ is implicitly a member of $K_w$. The substitution $\{x = \text{thesis.tex}, y = 30000\}$, produces a ground instance of $x > y$ and also satisfies conditions (1) and (2) of step 7. Because there are no other conjuncts, condition (3) is satisfied trivially. Thus, $\text{IA}(\text{size(thesis.tex)}) = \text{W}$. Intuitively this query captures the notion that since the agent will come to know the value of $\text{size(thesis.tex)}$, then it will also come to know whether or not this value is larger than 30000.

4.3 Proofs

In this section, a proof is given to show that the inference algorithm $\text{IA}$ is sound. An example illustrates that $\text{IA}$ is incomplete.

Before proving that $\text{IA}$ is sound, a preliminary result is proven. This result shows that $\text{EvalTerm}$ is correct, in the sense that the simplifications it makes to the input are
entailed by the agent’s knowledge.

**Lemma 4.3.1** Let \( \varepsilon \) be a formula input to \( \text{IA} \), and let \( \varepsilon' \) represent the formula resulting from the simplification of the terms in \( \varepsilon \) by \( \text{EvalTerm} \). Then the following relationships hold: (1) If \( \text{KB} \models K(\varepsilon') \) then \( \text{KB} \models K(\varepsilon) \), (2) If \( \text{KB} \models K(\neg \varepsilon') \) then \( \text{KB} \models K(\neg \varepsilon) \), (3) If \( \text{KB} \models K_{\text{whe}}(\varepsilon') \) then \( \text{KB} \models K_{\text{whe}}(\varepsilon) \), and (4) If \( \text{KB} \models \exists x.K(x = \varepsilon') \) then \( \text{KB} \models \exists x.K(x = \varepsilon) \).

**Proof:** If \( \varepsilon' \) is syntactically identical to \( \varepsilon \) then the lemma is trivially true. Assume instead that \( \varepsilon' \) is not syntactically identical to \( \varepsilon \). This means that procedure \( \text{EvalTerm} \) must have made at least one substitution to a term of \( \varepsilon \), or to an argument of one of its terms (or to an argument of an argument, etc.). Moreover, each substitution must have replaced a single function with constant arguments, \( f(\bar{c}) \), by a constant \( d \) (step 2 of \( \text{EvalTerm} \)). Define the sequence \( s_1, \ldots, s_n \) to be the ordered sequence of substitutions that were applied to \( \varepsilon \) by \( \text{EvalTerm} \) (there are only a finite number of terms in the original formula \( \varepsilon \) and the recursion bottoms out when it tries to simplify constant terms, so only a finite number of substitutions are made). Each \( s_i \) records the particular term that the substitution was applied to, and the substitution, \( f(\bar{c}_i) = d_i \), that was made.

Applying a substitution \( s_i \) to a formula \( \varepsilon_i \) transforms it syntactically into the formula \( \varepsilon_{i+1} \) where a specific term \( f(\bar{c}_i) \) in \( \varepsilon_i \) is replaced by \( d_i \). Define \( s_i^{-1} \) to be the inverse substitution, so that applying a substitution \( s_i^{-1} \) to the formula \( \varepsilon_{i+1} \) transforms it back into \( \varepsilon_i \). Applying the substitutions \( s_1, \ldots, s_n \) to \( \varepsilon \) has the effect of transforming it syntactically into \( \varepsilon' \). Similarly, the inverse substitution sequence \( s_n^{-1}, \ldots, s_1^{-1} \) transforms \( \varepsilon' \) syntactically into \( \varepsilon \):

\[
\varepsilon = \varepsilon_1 \xrightarrow{s_1} \varepsilon_2 \xrightarrow{s_2} \ldots \xrightarrow{s_{n-1}} \varepsilon_n \xrightarrow{s_n} \varepsilon_{n+1} = \varepsilon'
\]

\[
\varepsilon = \varepsilon_1 \xleftarrow{s_1^{-1}} \varepsilon_2 \xleftarrow{s_2^{-1}} \ldots \xleftarrow{s_{n-1}^{-1}} \varepsilon_n \xleftarrow{s_n^{-1}} \varepsilon_{n+1} = \varepsilon'
\]
Each substitution $s_i$ replaces a function term $f(c_i)$ with a constant $d_i$, with the requirement that $f(c_i) = d_i \in K_f$ (or it can be computed that $f(c_i) = d_i$, which means that this function mapping is implicitly in $K_f$). By the semantics of $K_f$, for each substitution, $f(c_i) = d_i \in K_f$, $KB \models K(f(c_i) = d_i)$.

Now assume that $KB \models K(\varepsilon')$. Since $s_n^{-1}$ transforms $\varepsilon'$ syntactically into $\varepsilon_n$ (by replacing a constant $d_n$ with the function $f(c_n)$), and $KB \models K(f(c_n) = d_n)$, then $KB \models K(\varepsilon_n)$. This process can now be repeated for each of the remaining substitutions $s_{n-1}^{-1}, \ldots, s_1^{-1}$ in turn, resulting in the conclusions $KB \models K(\varepsilon_{n-1}), \ldots, KB \models K(\varepsilon_1) \equiv KB \models K(\varepsilon)$.

This process can be repeated for the other three relationships, by applying the same “chain” of reasoning. For (3) it is important to note that $K_{wm}(\phi) := K(\phi) \lor K(\neg \phi)$. Thus, if $KB \models K(\varepsilon_i) \lor K(\neg \varepsilon_i)$ and $KB \models K(f(c_{i-1}) = d_{i-1})$, then it follows that $KB \models K(\varepsilon_{i-1}) \lor K(\neg \varepsilon_{i-1})$. ■

By using the results of the previous lemma, the proof that the inference algorithm $IA$ is sound can now be given.

**Theorem 4.3.2** The inference algorithm $IA$ is sound.

**Proof:** To show that $IA$ is sound, the output corresponding to a given input must be shown to be correct in all cases. For instance, if $IA(\varepsilon)$ returns $T$, then it must be shown that in fact $KB \models K(\varepsilon)$. As a result, the proof is divided into three parts, each part considering an output of $T$, $F$, and $W$ respectively.

$IA$ takes as input a formula $\varepsilon$, but its first step is to try to simplify the formula by using $EvalTerm$. The result is the formula $\varepsilon'$. The remainder of the algorithm works with $\varepsilon'$, however the output is given in term of $\varepsilon$. Lemma 4.3.1 allows the proof to be simplified by showing that a formula entailed from $KB$ involving $\varepsilon'$ implies that the same
formula involving $\varepsilon$ can be entailed from $\mathbf{KB}$ as well. Thus, it suffices to work solely with $\varepsilon'$ in this proof.

(i) Assume $\mathbf{IA}(\varepsilon') = \mathbf{T}$. Then either step 3 or 4 of $\mathbf{IA}$ must have been applied. If step 4 was applied then $\varepsilon' \in K_f$. By the semantics of $K_f$, $\mathbf{KB}$ includes the formula $K(\varepsilon')$ (Equation 3.3). Thus $\mathbf{KB} \models K(\varepsilon')$. If step 3 was applied then $\varepsilon'$ must be of the form $t_1 = t_2$, and the two terms, $t_1$ and $t_2$, are syntactically identical. Identical terms must have the same denotation in any single world (although that denotation can vary from world to world). That is, in every world the formula $t_1 = t_2$ is satisfied. Thus, by the semantics of $K$ we have that $\mathbf{KB} \models K(t_1 = t_2)$ and so $\mathbf{KB} \models K(\varepsilon')$.

(ii) Assume $\mathbf{IA}(\varepsilon') = \mathbf{F}$. Then either step 3, 5, or 6 of $\mathbf{IA}$ must have been applied. If step 5 was applied, then $\neg \varepsilon' \in K_f$. By the semantics of $K_f$, $\mathbf{KB}$ includes the formula $K(\neg \varepsilon')$. Thus, $\mathbf{KB} \models K(\neg \varepsilon')$. If step 3 was applied, then we have the case that $\varepsilon'$ is of the form $t_1 = t_2$, and that $t_1$ and $t_2$ are both constants. However, $t_1$ and $t_2$ are not syntactically identical, otherwise $\mathbf{IA}$ would have returned $\mathbf{T}$ at step 3. Since $t_1$ and $t_2$ are syntactically distinct constants, then by Equation 3.2 $\mathbf{KB}$ includes the formula $K(t_1 \neq t_2)$. In other words, $\mathbf{KB} \models K(\neg \varepsilon')$.

If step 6 was applied, then there exists $\phi(\bar{x}) = \alpha_1(\bar{x}) \land \ldots \land \alpha_k(\bar{x}) \in LCW$ and a ground instance of $\phi$, $\phi(\bar{x}/\bar{a})$, such that (1) $\bar{a}$ are constants appearing in $K_f$, (2) $\alpha_i(\bar{x}/\bar{a}) = \varepsilon'$ for some $i$, and (3) $\mathbf{IA}(\alpha_j(\bar{x}/\bar{a})) = \mathbf{T}$ for all $j \neq i$. Since $\phi(\bar{x}) \in LCW$, then $\mathbf{KB}$ includes the formula:

$$\forall \bar{x}. \bigwedge_{\varepsilon \in C} \neg(x_1 = c_1 \land \ldots \land x_n = c_n) \Rightarrow K(\neg \phi(\bar{x})) \quad (4.1)$$

where, $C = \{ \varepsilon : \alpha_i(\bar{x}/\varepsilon') \in K_f, 1 \leq i \leq k \}$. Since by (2), $\alpha_i(\bar{x}/\bar{a}) = \varepsilon'$, we have that $\alpha_i(\bar{x}/\bar{a}) \notin K_f$, otherwise $\mathbf{IA}(\varepsilon')$ would have returned $\mathbf{T}$. This means that $\bar{a} \notin C$. Thus,
from Formula 4.1 we can reason that for the substitution \{x/\bar{a}\} we have:

\[
\begin{align*}
\text{KB} & \models K(\neg \phi(x/\bar{a})) \\
& \models K(\neg (\alpha_1(x/\bar{a}) \land \ldots \land \alpha_k(x/\bar{a}))) \\
& \models K(\neg \alpha_1(x/\bar{a}) \lor \ldots \lor \neg \alpha_k(x/\bar{a}))
\end{align*}
\] (4.2)

The universal quantification disappears since \phi(x/\bar{a}) is a ground instance of \phi. By (3), since \text{IA}(\alpha_j(x/\bar{a})) = T for all \(j \neq i\) then \(\text{KB} \models K(\alpha_j(x/\bar{a}))\) for all \(j \neq i\) (from the result in part (i) of the proof). From Formula 4.2 we have that:

\[
\begin{align*}
\text{KB} & \models K(\neg \alpha_1(x/\bar{a}) \lor \ldots \lor \neg \alpha_k(x/\bar{a})) \land \\
& K(\alpha_j(x/\bar{a}))
\end{align*}
\] (4.3)

By semantics of \(K\) we have that \(K(\varphi) \land K(\psi) \Rightarrow K(\varphi \land \psi)\). That is, conjunctions of formulas involving \(K\) operators can be brought within a single \(K\) operator. Applying this property to Formula 4.3 results in the formula:

\[
\begin{align*}
\text{KB} & \models K((\neg \alpha_1(x/\bar{a}) \lor \ldots \lor \neg \alpha_k(x/\bar{a})) \land \\
& \alpha_1(x/\bar{a}) \land \ldots \land \alpha_{i-1}(x/\bar{a}) \land \alpha_{i+1}(x/\bar{a}) \land \ldots \land \alpha_k(x/\bar{a}))
\end{align*}
\] (4.4)

As well, we also know that \(\varphi \iff \psi\) entails \(K(\varphi) \iff K(\psi)\). That is, logical simplifications can be performed within the \(K\) operator. Simplifying Formula 4.4 results in the formula \(\text{KB} \models K(\neg \alpha_i(x/\bar{a}))\). That is, \(\text{KB} \models K(\neg \varepsilon')\).

(iii) Assume \(\text{IA}(\varepsilon') = W\). Then either step 2 or 7 of \(\text{IA}\) was applied. If step 2 was applied, then there are two cases to consider. In the first situation, \(\varepsilon'\) is the term \(t\) and \(t\) is a constant. Since constants are rigid then \(\text{KB}\) includes the formula \(\exists x.K(x = t)\) (Equation 3.1). In other words, \(\text{KB} \models \exists x.K(x = \varepsilon')\). In the second situation \(\varepsilon'\) is the
term \( t \), and there exists a \( t' \in K_v \) and substitution \( \theta \) such that \( t'\theta = t \). Since \( t' = t'(\bar{x}) \in K_v \), \( KB \) includes the formula \( \forall \bar{x}.\exists v.K(t'(\bar{x}) = v) \). Thus, \( KB \models \exists v.K(t' = v) \) since this is an instantiation of the universal. Since \( t' = e' \) we have that \( KB \models \exists v.K(e' = v) \).

If step 7 was applied, then there exists a \( \phi(\bar{x}) = \alpha_1(\bar{x}) \land \ldots \land \alpha_k(\bar{x}) \in K_w \) and a ground instance of \( \phi \), \( \phi(\bar{x}/\bar{a}) \), such that (1) \( \bar{a} \) are either constants appearing in \( K_f \) or terms \( t_i \) appearing in \( e' \) for which \( IA(t_i) = W \), (2) \( \alpha_i(\bar{x}/\bar{a}) = e' \) for some \( i \), and (3) \( IA(\alpha_j(\bar{x}/\bar{a})) = T \) for all \( j \neq i \). Since \( \phi(\bar{x}) \in K_w \) then:

\[
KB \models \forall \bar{x}.K(\alpha_1(\bar{x}) \land \ldots \land \alpha_k(\bar{x})) \lor K(\neg(\alpha_1(\bar{x}) \land \ldots \land \alpha_k(\bar{x}))) \tag{4.5}
\]

In particular, since \( \{\bar{x}/\bar{a}\} \) is a substitution such that \( \phi(\bar{x}/\bar{a}) \) is a ground instance of \( \phi \), then we can infer from Formula 4.5 that:

\[
KB \models K(\alpha_1(\bar{x}/\bar{a}) \land \ldots \land \alpha_k(\bar{x}/\bar{a})) \lor K(\neg(\alpha_1(\bar{x}/\bar{a}) \land \ldots \land \alpha_k(\bar{x}/\bar{a}))) \tag{4.6}
\]

From (3) it is given that \( IA(\alpha_j(\bar{x}/\bar{a})) = T \) for all \( j \neq i \). Thus, \( KB \models K(\alpha_j(\bar{x}/\bar{a})) \) for all \( j \neq i \) (from the result in part (i) of the proof). Conjoining this result with Formula 4.6 results in:

\[
KB \models K(\alpha_1(\bar{x}/\bar{a}) \land \ldots \land \alpha_k(\bar{x}/\bar{a})) \lor K(\neg(\alpha_1(\bar{x}/\bar{a}) \land \ldots \land \alpha_k(\bar{x}/\bar{a}))) \land K(\neg(\alpha_1(\bar{x}/\bar{a}) \land \ldots \land \alpha_k(\bar{x}/\bar{a}))) \tag{4.7}
\]

Moving the conjuncts into a single \( K \) operator on each side of the disjunction, and sim-
plifying both disjuncts results in the formula:

\[
\text{KB} \models K(\alpha_1(x/a)) \lor K(\neg\alpha_1(x/a))
\]  

(4.8)

This means that \(\text{KB} \models K(\varepsilon') \lor K(\neg\varepsilon')\), or in other words that \(\text{KB} \models K_{\text{whale}}(\varepsilon')\) (since \(\varepsilon'\) is a ground atomic formula).

Even though the inference algorithm is sound, it is not complete. That is, there are formulas that are logically entailed by an agent’s knowledge state, that cannot be derived by using IA. This is not surprising since IA is restricted to queries of ground atomic formulas or variable-free terms. The following example illustrates one of the limitations of IA.

Consider the case where \(p(a) \land q(a, x) \in K_w\) and \(p(a) \in K_f\). By the semantics of \(K_w\) and \(K_f\), \(\text{KB} \models \forall x.K(p(a) \land q(a, x)) \lor K(\neg(p(a) \land q(a, x)))\) and \(\text{KB} \models K(p(a))\). Thus,

\[
\text{KB} \models [\forall x.K(p(a) \land q(a, x)) \lor K(\neg(p(a) \land q(a, x)))] \land K(p(a))
\]

\[
\models [\forall x.K(p(a) \land q(a, x)) \land K(p(a))] \lor [\forall x.K(\neg p(a) \lor \neg q(a, x)) \land K(p(a))]
\]

\[
\models [\forall x.K(p(a)) \land K(q(a, x))] \lor [\forall x.K(\neg p(a) \lor \neg q(a, x)) \land K(p(a))]
\]  

(4.9)

Moving the conjuncts on both sides of the disjunction into a single \(K\) operator and simplifying results in the formula:

\[
\text{KB} \models \forall x.K(q(a, x)) \lor K(\neg q(a, x)).
\]

Thus \(\text{KB}\) entails “know whether” knowledge of the formula \(q(a, x)\). The inference algorithm IA is able to answer queries about instances of the formula \(q(a, x)\). For
instance, a query such as $IA(q(a,c))$ would return an answer of $W$, indicating that $KB \models K_{whe}(q(a,c))$. However, $IA$ is not able to infer the universal $\forall x.K_w(q(a,x))$.\footnote{This is not a significant problem in practice. Whereas such queries might be useful, in a planning situation the type of inferences that are typically needed are queries about ground formulas.}

The inference algorithm is stated as a recursive algorithm. The complexity of $IA$ is dominated by steps 6 and 7, where the algorithm is recursively called to search for ground instances of the formula $\phi(\vec{x})$. For example, consider a formula $p(x) \land q(x,y)$ in $LCW$, and the query $IA(p(a))$. The inference algorithm may have to consider many ground instances of the conjunct $q(a,y)$ (for instance, $q(a,b)$, $q(a,d)$, etc.) before it can conclude anything about $p(a)$. The number of ground instances of $\phi(\vec{x})$ can potentially be exponential in the number of variables of $\vec{x}$.

### 4.4 Primitive Queries

The inference procedure $IA$ takes as input a query about a ground atomic formula or variable free term. Some queries, however, such as those involving negation, cannot be made directly using $IA$. Also, there are certain types of queries that are useful in a planning situation and are used quite regularly, such as testing if an agent knows whether some formula or its negation is true. As a result, a simple query language is developed in this section that defines a set of primitive queries, that can be used to describe queries such as the ones mentioned above.

A primitive query relates a knowledge condition to the results of a query made with the inference algorithm. To satisfy a knowledge condition about some formula, a query to the inference algorithm about a corresponding formula must return a certain output, this output indicating that the knowledge condition is true. For example, satisfying the knowledge query that $K'(\neg \alpha)$ is true means that the inference algorithm invoked with...
4.4. PRIMITIVE QUERIES

1. \( K(\alpha), \) true if \( IA(\alpha) \) returns \( T. \)
2. \( K(\neg \alpha), \) true if \( IA(\alpha) \) returns \( F. \)
3. \( K_w(\alpha) \) or \( K_w(\neg \alpha), \) true if \( IA(\alpha) \) returns \( W, \) \( T, \) or \( F. \)
4. \( K_v(t), \) true if \( IA(t) \) returns \( W. \)
5. \( \neg K(\alpha), \) true if \( IA(\alpha) \) returns \( F \) or \( U. \)
6. \( \neg K(\neg \alpha), \) true if \( IA(\alpha) \) returns \( T \) or \( U. \)
7. \( \neg K_w(\alpha) \) or \( \neg K_w(\neg \alpha), \) true if \( IA(\alpha) \) returns \( U. \)
8. \( \neg K_v(t), \) true if \( IA(t) \) returns \( U. \)

| Table 4.2: Primitive queries |

\( IA(\alpha) \) must return \( F. \) An output of \( F \) means that \( KB \models K(\neg \alpha) \) so the knowledge condition is satisfied. Primitive queries allow an agent to make specific types of tests about its knowledge, without needing to know in general how to translate such a test into an inference algorithm query, or how to interpret the results of such a query. The types of primitive queries that can be made are listed in Table 4.2. In this listing, \( \alpha \) represents any ground atomic formula, and \( t \) is any variable free term.

Primitive queries can be made for determining if an agent knows the truth of a formula, if an agent knows some function value, or if an agent knows whether some formula or its negation is true. For instance, the condition \( K(\alpha) \) is satisfied as true if \( IA(\alpha) \) returns \( T. \) Determining the truth of \( K(\neg \alpha) \) is similar, except the query \( IA(\alpha) \) must return \( F. \) To determine the truth of a \( K_w(\alpha) \) condition, \( IA(\alpha) \) can return value of \( W, \) \( T, \) or \( F. \) A return value of \( W \) implies \( KB \models K_w(\alpha), \) so the condition is obviously satisfied. If \( IA \) returns \( T \) then \( KB \models K(\alpha) \) which implies that \( KB \models K_w(\alpha). \) Similar reasoning
is used if \( \text{IA} \) returns \( \mathbf{F} \). Satisfying the knowledge condition \( K_w(\neg \alpha) \) is the same as satisfying the knowledge condition \( K_w(\alpha) \). If it can be shown that \( \mathbf{KB} \models K_{\text{whe}}(\alpha) \), this implies that \( \mathbf{KB} \models K_{\text{whe}}(\neg \alpha) \) since \( K_{\text{whe}}(\alpha) = K_{\text{whe}}(\neg \alpha) \). The knowledge condition \( K_v(t) \) takes a term \( t \) as its argument. \( \text{IA} \) can only return \( \mathbf{W} \) to satisfy the query, indicating that \( \mathbf{KB} \models \exists x. K(x = t) \) (the only other result that \( \text{IA} \) can produce with an input of a single term is \( \mathbf{U} \)).

The negation of the above queries can also be made.\(^4\) The formula \( \neg K(\alpha) \) means semantically that \( \neg \alpha \) is considered possible by the agent.\(^5\) This knowledge condition is satisfied if the query \( \text{IA}(\alpha) \) returns \( \mathbf{F} \), \( \mathbf{W} \), or \( \mathbf{U} \). In each case, the agent considers \( \neg \alpha \) possible. For instance, if \( \mathbf{F} \) is returned, then \( \mathbf{KB} \models K(\neg \alpha) \), so \( \neg \alpha \) holds in all worlds considered possible. If \( \mathbf{W} \) is returned, \( \mathbf{KB} \models K_{\text{whe}}(\alpha) \), so either all worlds will have \( \alpha \) hold or \( \neg \alpha \) hold. Thus \( \neg \alpha \) is possible. If \( \mathbf{U} \) is returned, nothing can be determined about \( \alpha \), so the agent may assume that both \( \alpha \) and \( \neg \alpha \) are possible. Similarly, satisfying the knowledge condition \( \neg K(\neg \alpha) \) means showing that \( \alpha \) is considered possible. So a return value of \( \mathbf{T} \), \( \mathbf{W} \), or \( \mathbf{U} \) on a query of \( \text{IA}(\alpha) \) is required. Consider the knowledge statement \( \neg K_w(\alpha) \). Semantically, this can be expressed as the agent “doesn’t know whether \( \alpha \)”. This means that the agent considers both \( \alpha \) and \( \neg \alpha \) possible. In this case, the only way to satisfy this knowledge condition is if \( \text{IA}(\alpha) \) returns \( \mathbf{U} \). Knowing either \( \alpha \) or \( \neg \alpha \) implies that the other cannot be known. However, when \( \mathbf{U} \) is returned, the inference algorithm is unable to determine anything about \( \alpha \) so the agent can consider both \( \alpha \) and \( \neg \alpha \) possible. The last negated condition, \( \neg K_v(t) \) can only be satisfied if \( \text{IA}(t) \) returns \( \mathbf{U} \), indicating that the agent has no information about the formula and so does not know its value.

\(^4\)The soundness of primitive queries 1–4 in Table 4.2 follows directly from the soundness proof of the inference algorithm. The soundness of the negative primitive queries (queries 5–8 in Table 4.2) has not yet been proved. Doing so requires only knowing extensions [Lev90].

\(^5\)Possibility can be viewed as the dual of knowledge. See [FHMV95].
Chapter 5

Representing Actions

The previous two chapters described a framework for representing an agent’s knowledge state in a STRIPS like manner as a collection of databases. In this chapter, a STRIPS like approach to modelling an agent’s actions is developed.

5.1 Extending the STRIPS Action Representation

Recall from Chapter 1 that a STRIPS action was specified by its parameters, preconditions, add list and delete list. In particular, the preconditions specified a list of conditions that must be true in the database \( D \) for the action to be applied. The effects of the action described updates to the database \( D \), by specifying the literals to be added (the add list), and the literals to be removed from \( D \) (the delete list). Updating the database \( D \) has the effect of transforming the world state represented by \( D \) to a new world state.

In this thesis, however, an agent’s knowledge is modelled by a collection of databases, \( DB \). As a result, the STRIPS action specification is extended to support interaction with multiple databases. For instance, satisfying the precondition of an action involves testing the contents of the various databases. An action’s effects are extended to update specific
databases but are still fundamentally described by a set of primitive add and delete operations. Thus, starting with an initial configuration of databases that represent the agent’s knowledge state, it is possible to decide what actions can be applied, and then generate what the new knowledge state will be after the action has been applied. The action representation in this thesis however goes further than the STRIPS representation by adopting some ADL style structures as extensions [Ped89].

Throughout this thesis, the separation of plan time from execution time has been an important theme. This separation is apparent in the action specification, where every action has a set of plan time effects and a set of execution time effects. Both sets of effects, however, are encoded as database updates. Thus, the same formalism can be used to compute the plan time effects of a sequence of actions, and track their execution time effects as well. An example of how an action affects an agent’s knowledge state at plan time and at execution time is given later in the chapter. More complex examples that illustrate how to make use of this formalism to reason about sequences of actions will be presented in Chapter 7. The rest of this chapter describes the formal representation of actions.

5.2 Action Specification

Actions are specified by four components: the parameters, the preconditions, the plan time effects, and the execution time effects.

Parameters: An action’s parameters are a set of variables that can be bound to produce a particular instance of the action. The parameters specified in this declaration can be used throughout the action description when specifying formulas for an action’s preconditions or effects.

Preconditions: The agent that is planning or executing a sequence of actions does not
have direct access to the state of its environment. Deciding whether or not an action can be applied depends solely on the agent’s knowledge state. Thus, in order to determine if the action’s preconditions are satisfied, the status of the databases must be queried. To accomplish this, the query language developed in Section 4.4 is used. Preconditions are specified as a conjunctive list of the primitive queries in Table 4.2. The primitive queries avoid making reference to specific databases in the actual precondition declaration, but instead utilize the inference algorithm $\text{IA}$. The inference algorithm reasons about the collection of databases to determine the truth of any formula in question. For an action’s precondition to be satisfied, all the queries in the list must evaluate to true. Besides specifying primitive queries with ground atomic formulas or variable free terms, the formulas in a primitive query can also make use of variables from the parameter list, these variables being bound to objects when the action is instantiated.

**Plan Time Effects:** An action’s plan time effects are specified by a list of condition-effect statements of the form $C \Rightarrow E$. A condition, $C$, is specified as a conjunctive set of primitive queries. All the queries of the particular condition must evaluate to true for the corresponding effect to be applied. However, the conditions are optional, so a simple list of effects may be given instead. An effect, $E$, specifies a set of additions or deletions to the four databases (see Section 5.3). Conditions and effects can make use of the variables from the parameter declaration.

**Execution Time Effects:** The execution time effects specify the interface between the execution module and the planner. The specification begins with a declaration of the name of the action and a list of run-time variables [GW96]. When the action instance is executed, this information is passed to the execution module. The execution module binds the run-time variables with information it gathers from the environment as a result of the action being executed.\(^1\) 

\(^1\)The order that the run-time variables are specified is important (they are positional, similar to passing
sequence of bindings for a subset (possibly all) of the run-time variables, depending on the action and the environment. The effects of the action are specified by condition-effect statements. Each condition-effect statement is of the form \( C \Rightarrow E \), similar to the plan-time effects specifications (including the use of variables from the parameter declaration) except that for execution time effects, \( C \) and \( E \) can make use of any of the run-time variables. As well, \( C \) may contain tests on the run-time variables, allowing conditions to be based on execution time values returned from the environment. As with the plan time effects, the condition list is optional. Since the execution module may return a sequence of bindings for particular run-time variables, a number of consecutive condition-effect statements may be enclosed by braces \{ \ldots \} to indicate that those statements should be applied to each distinct binding of the run-time variables.\(^2\) The remainder of an action’s condition-effect statements (those outside the \{ \ldots \}) are each considered only once. These statements specify the execution time effects that do not contain run-time variables, the statements that make use of the “current” bindings of run-time variables, or can be used to specify condition-effect statements with run-time variables that only return a single set of bindings.

A formal syntax for specifying an action is provided by the grammar given in Table 5.1. The grammar symbols surrounded by square braces \{ \ldots \} indicate that they are optional (for instance [param-list] indicates that a parameter list does not necessarily need to be specified). Terminal symbols that must be included syntactically as shown are given in boldface.

---

\(^2\) This is similar to a program’s “loop” structure, in that the statements contained within the \{ \ldots \} specify the effects that are repeated for each instance of run-time variables. At the end of each loop, the run-time variables are “initialized” with the next binding in the sequence and the statements within the braces are reconsidered.
5.2. ACTION SPECIFICATION

\[
\text{action} \quad \rightarrow \quad \text{action-name \ [param-list]}
\]

**Preconditions:** [preconds]

**Plan-time Effects:** [plan-effects]

**Execution-time Effects:** [exec-spec]

\[
\text{param-list} \quad \rightarrow \quad \text{var} \mid \text{var, param-list}
\]

\[
\text{preconds} \quad \rightarrow \quad \text{primitive-query} \mid \neg\text{primitive-query} \mid \text{preconds, preconds}
\]

\[
\text{primitive-query} \quad \rightarrow \quad K(\alpha) \mid K(\neg\alpha) \mid K_w(\alpha) \mid K_w(\neg\alpha) \mid K_v(t)
\]

\[
\text{plan-effects} \quad \rightarrow \quad \text{effect} \mid \text{effect plan-effects}
\]

\[
\text{effect} \quad \rightarrow \quad \text{update-ops} \mid \text{preconds} \Rightarrow \text{update-ops}
\]

\[
\text{update-ops} \quad \rightarrow \quad \text{add}(db, \varepsilon_{db}) \mid \text{del}(db, \varepsilon_{db}) \mid \text{update-ops, update-ops}
\]

\[
\text{exec-spec} \quad \rightarrow \quad \text{exec(action-exec, rtvar-list)}
\]

\[
\text{exec-list} \quad \rightarrow \quad \text{exec-list} \mid \{\text{exec-list}\} \mid \text{exec-list}
\]

\[
\text{action-exec} \quad \rightarrow \quad \text{name of action to be executed, possibly specified using variables from param-list}
\]

\[
\text{rtvar-list} \quad \rightarrow \quad \text{rtvar} \mid \text{rtvar, rtvar-list}
\]

\[
\text{exec-effect} \quad \rightarrow \quad \text{exec-effect} \mid \text{exec-effect exec-list}
\]

\[
\text{exec-cond} \quad \rightarrow \quad \text{exec-cond} \Rightarrow \text{update-exec}
\]

\[
\text{update-exec} \quad \rightarrow \quad \text{add}(db, \varepsilon_{db}) \mid \text{del}(db, \varepsilon_{db}) \mid \text{update-exec, update-exec}
\]

\[
\text{exec-query} \quad \rightarrow \quad \text{exec-query} \mid \text{exec-query, exec-cond}
\]

\[
\text{primexec-query} \quad \rightarrow \quad K(\alpha) \mid K(\neg\alpha) \mid K_w(\alpha) \mid K_w(\neg\alpha) \mid K_v(t)
\]

\[
\text{db} \quad \rightarrow \quad K_f \mid K_w \mid K_v \mid \text{LCW}
\]

Where:

- \(\varepsilon_{db}\) is a formula accepted by \(db\) allowing variables from \(param-list\),
- \(\text{var}\) is a variable,
- \(\text{rtvar}\) is a run-time variable,
- \(\text{rtvar-test}\) is a test of a run-time variable’s value,
- \(\alpha\) is a ground atomic formula, or an atomic formula with variables from \(param-list\),
- \(t\) is a variable free term, or a term with variables from \(param-list\),
- * indicates that the formula can make use of run-time variables from \(rtvar-list\)

Table 5.1: Action specification grammar
5.3 Specifying Database Additions and Deletions

Additions and deletions to the four databases are specified by the two primitive operations, \textit{add} and \textit{delete}, which take as arguments the database to perform the operation on, and a formula. Thus, a formula like \textit{add}(K_w, \text{in-dir}(x, /project) \land \text{size}(x) = y) indicates that the conjunction should be added to the $K_w$ database, and a formula like \textit{delete}(K_f, \text{readable}(\text{thesis.tex})) indicates that the literal should be removed from the $K_f$ database.

Algorithms for the \textit{add} and \textit{delete} primitives are given in Table 5.2. In particular, the primitives are configured so as to maintain the two consistency conditions required in Theorem 3.5.1. If a fact is added to $K_f$ whose negation is also a fact in $K_f$, then the negation is deleted. When functions are added to $K_f$ any previous function values are deleted. These algorithms also contain two small “optimizations”. If a function is added to $K_f$ then that function is deleted from $K_v$ if present. Also, if a function mapping is being deleted from $K_f$, the \textit{delete} procedure is configured to allow the user to simply specify the “function” part without knowing the specific mapping, for instance \textit{delete}(K_f, f(c_1, \ldots, c_k)).

5.4 Example: Applying an Action to a Knowledge State

In this section we look at an example of an agent applying an action, both at plan time and execution time. Consider an agent in the UNIX domain that wishes to count the number of words in the file \textit{thesis.tex}. This can be achieved by using the UNIX command \textit{wc} that takes a single argument, the file to perform the word count on. The action specification for the UNIX command \textit{wc} is shown in Table 5.3. The initial knowledge state of the agent, $S$, is described by the database $K_f = \{ \text{readable(thesis.tex)} \}$, with the other
5.4. EXAMPLE: APPLYING AN ACTION TO A KNOWLEDGE STATE

Procedure \textit{add}(db, \varepsilon)

\textbf{Inputs:} a database \((K_f, K_v, K_w, LCW)\) represented by \(db\), and a formula \(\varepsilon\) of a form accepted by \(db\)

1. If \(db = K_f\) and \(-\varepsilon \in K_f\) then delete\((K_f, -\varepsilon)\).

2. If \(db = K_f\) and \(\varepsilon\) is a function of the form \(f(c_1, \ldots, c_k) = c_{k+1}\), and there exists \(f(c_1, \ldots, c_k) = d \in K_f, c_{k+1} \neq d\), then delete\((K_f, f(c_1, \ldots, c_k))\).

3. If \(db = K_f\) and \(\varepsilon\) is a function of the form \(f(c_1, \ldots, c_k) = c_{k+1}\), and there exists \(f(c_1, \ldots, c_k) \in K_v\), then delete\((K_v, f(c_1, \ldots, c_k))\).

4. If \(\varepsilon \notin db\) then add \(\varepsilon\) to \(db\).

Procedure \textit{delete}(db, \varepsilon)

\textbf{Inputs:} a database \((K_f, K_v, K_w, LCW)\) represented by \(db\), and a formula \(\varepsilon\)

1. If \(\varepsilon = f(c_1, \ldots, c_k)\) and \(db = K_f\), then remove all formulas of the form \(f(c_1, \ldots, c_k) = c_{k+1}\) from \(K_f\).

2. If \(\varepsilon \in db\) then remove \(\varepsilon\) from \(db\).

Table 5.2: Database \textit{add} and \textit{delete} primitives

databases being empty.

First, we consider the agent reasoning about the action \textit{wc thesis.tex} at plan time. The action specification for the \textit{wc} action has a single parameter, \(x\). Since the action is being applied to the file \textit{thesis.tex}, \(x\) is bound to \textit{thesis.tex} to produce a specific instance of the action. The single precondition, \(K(\text{readable}(x))\), is resolved to \(K(\text{readable(thesis.tex)})\) as a result. Satisfying this precondition means satisfying the primitive query \(K(\text{readable(thesis.tex)})\) which is true if \textbf{IA(\text{readable(thesis.tex)})} returns \(T\). Since \textit{readable(thesis.tex)} is in \(K_f\), the inference algorithm does in fact return \(T\) (step 4 of \textbf{IA}). The function \textit{wordcount(thesis.tex)} is added into the \(K_v\) database as a result of the action’s single plan time \textit{add} operation. Having the function \textit{wordcount(thesis.tex)} in \(K_v\) does not mean that the agent knows the actual number of words in the \textit{thesis.tex}. 


Instead it indicates that the agent will come to know this information. At this point the action has been successfully applied to the knowledge state of the agent, bringing about a new knowledge state, $S'$, represented by the databases $K_f = \{ \text{readable(thesis.tex)} \}$ and $K_v = \{ \text{wordcount(thesis.tex)} \}$, with the other databases empty.

Now consider the agent executing the action $\text{wc thesis.tex}$, from the initial knowledge state $S$. Again $\text{thesis.tex}$ is bound to the parameter $x$ to produce a particular instance of the action. As with the plan time situation, the preconditions of the action are satisfied and the execution time effects can be applied. With $x$ bound to $\text{thesis.tex}$, the directive $\text{exec(\text{wc thesis.tex}, !\text{words})}$ is passed to the execution module. This indicates that the execution module should execute the UNIX command $\text{wc thesis.tex}$, and that the results should be bound to the run-time variable $!\text{words}$. Running the command in the UNIX environment returns a value of 20000 that is bound to $!\text{words}$. Applying the $\text{add}$ operation now has the effect of adding the function mapping $\text{wordcount(thesis.tex)} = 20000$ to $K_f$. In this case the $\text{delete}$ operation does nothing since $\text{wordcount(thesis.tex)}$ was not in $K_v$. The new knowledge state of the agent, $S'$, is now represented by $K_f = \{ \text{readable(thesis.tex)}, \text{wordcount(thesis.tex)} = 20000 \}$, with the other databases remaining empty. Figure 5.1 illustrates both the plan time and execution time results of applying the $\text{wc}$ action to a knowledge state.

This example shows how the action specification, inference algorithm, and the primi-
5.4. **EXAMPLE: APPLYING AN ACTION TO A KNOWLEDGE STATE**

Plan Time

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K_f)</td>
<td>readable(thesis.tex)</td>
</tr>
</tbody>
</table>

\[ \text{wc thesis.tex} \]

Initial Knowledge State

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K_f)</td>
<td>readable(thesis.tex)</td>
</tr>
</tbody>
</table>

Execution Time

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K_f)</td>
<td>readable(thesis.tex)</td>
</tr>
</tbody>
</table>

\[ \text{wc thesis.tex} \]

!words = 20000

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K_f)</td>
<td>readable(thesis.tex)</td>
</tr>
<tr>
<td>(K_v)</td>
<td>wordcount(thesis.tex) = 20000</td>
</tr>
</tbody>
</table>

Initial Knowledge State

Figure 5.1: Applying the \text{wc thesis.tex} action

database updates interact when an action is applied. It also illustrates that an action’s plan time effects are not necessarily the same as its execution time effects. This is important for a planning agent reasoning about a sequence of actions. The more complete an agent’s knowledge is at plan time, the better prepared that agent is to reason about future actions, and to successfully build plans that achieve its goals. In Chapter 7, examples of action sequences from a variety of domains will be looked at, and the difference between the resulting plan time and execution time knowledge states will be illustrated.
Chapter 6

Handling Exceptions

When an agent’s knowledge state changes due to the effects of an action, there is a possibility that knowledge of one type may have an interaction with knowledge of a different type. In this chapter, the effects that an action has on the $K_f$ and $K_v$ databases are characterized, and used for discussing possible conflicts with the $K_w$ and $LCW$ databases. The notion of an exception is developed as a way of managing these conflicts. A simple procedure for handling exceptions is given, along with a description of the changes that are necessary to the existing algorithms, to support this new procedure.

The procedure described in this chapter for handling exceptions is presented as an extension, as opposed to a primary component, of the knowledge representation framework. Exception handling represents one possible starting point for addressing some of the problems that arise when an agent’s knowledge changes through action. The procedure is purposely conservative, but its development illustrates some of the issues that are involved.
6.1 Knowledge Updates and Action

An action that has the effect of adding or deleting formulas from a database in DB has the potential of interacting with other types of knowledge. For instance, consider the following situations:

- An agent’s LCW database contains the formula \( \text{in-dir}(x, /\text{project}) \land \text{readable}(x) \), indicating that the agent has LCW knowledge of the readability of all the files in the directory /project. The UNIX command \texttt{mv thesis.tex /project} has the execution time effect of adding the fact \( \text{in-dir}(\text{thesis.tex}, /\text{project}) \) to \( K_f \). However, no information is known as to whether or not \( \text{thesis.tex} \) is readable. The semantics of LCW reasoning however, now allow \( \neg \text{readable}(\text{thesis.tex}) \) to be inferred, which may not be true.

- An agent’s \( K_f \) database contains the literal \( \text{in-dir}(\text{thesis.tex}, /\text{project}) \). Its \( K_w \) database also contains the formula \( \text{in-dir}(x, /\text{project}) \land \text{size}(x) = y \), indicating that the agent will come to know the size of all the files in the directory /project. The other databases are empty. The UNIX command \texttt{compress thesis.tex} has the plan time effect of removing \( \text{size}(\text{thesis.tex}) \) from \( K_v \), making the size of this file unknown. There is no formula in \( K_v \) so nothing is removed. However, the semantics of reasoning with \( K_w \) at this point still imply that the size of \( \text{thesis.tex} \) will become known to the agent, even though this information should be unknown.

These two examples illustrate some of the possible conflicts that can occur between different types of knowledge. Even though the knowledge state is still consistent in the sense of Theorem 3.5.1, there are still concerns about the reliability of the knowledge state that can arise from updating an agent’s knowledge.
6.1. KNOWLEDGE UPDATES AND ACTION

An action’s effects are described by its sequence of database updates. An action \( a \) specifies a sequence of primitive database update operations \( e_1, \ldots, e_n \) (be it at plan time or at execution time), where \( e_i \) indicates an \textit{add} or \textit{delete} operation, a database, and a formula. This chapter primarily focuses on the types of changes that an action makes to the \( K_f \) and \( K_v \) databases, and the possible conflicts that changes to these databases can cause to \( K_w \) and LCW knowledge.

The problem stems from the fact that \( K_w \) and LCW formulas make assertions that are universally quantified over a set of objects. Formulas that are added or removed from \( K_f \) for instance, may match particular conjuncts of a \( K_w \) or LCW formula. As a result, the semantics of the universally quantified formulas may imply conditions that are not true. Consider the first example situation where the formula \( \text{in-dir}(x, /project) \land \text{readable}(x) \) is in LCW. Adding the fact \( \text{in-dir}(thesis.tex, /project) \) to \( K_f \) results in a situation where the fact \( \neg \text{readable}(thesis.tex) \) can be inferred. This is not necessarily true since the agent has no information about whether or not \( thesis.tex \) is readable and the action of moving a file into the directory /project doesn’t make it true.

The reason that a conflict occurs in the above situation is that the addition of a new fact to \( K_f \) adds a new “item” into the domain of the local closed world formula, an item that might not satisfy the LCW formula. Thus, reasoning with LCW results in incorrect conclusions, since the semantics of the LCW formula and the inference procedure still assume that the list of items in \( K_f \) satisfying the formula is complete.

If an entire “instance” of an LCW or \( K_w \) formula is added to the database collection, then this problem can possibly be avoided. For example, in the first situation, if \( \text{in-dir}(thesis.tex, /project) \) was added to \( K_f \) and \( \text{readable}(thesis.tex) \) was already known to be in \( K_f \), then the conflict would not have occurred. However, detecting if an instance of a LCW or \( K_w \) formula is completely satisfied, may not in general be computationally efficient. Consider the situation where the formula \( p(x) \land q(x, y) \) is in LCW. If the fact...
was added to $K_f$ then one would have to check to see if the second conjunct is completely satisfied. But this would mean checking to see if the facts $q(a, y)$ for all $y$ were known to the agent.

Another conflict occurs in the second example, when information from $K_u$ is deleted. Even though a formula isn’t explicitly listed in $K_u$ (or $K_f$) as in the above case, a conflict still occurs because the intent of the operation was to remove the formula, making it unknown to the agent. Thus, in this situation, an instance of the $K_w$ formula is implicitly incomplete (the size of the file thesis.tex becomes unknown). However, the $K_w$ formula still implies that the agent will come to know the size of all the files in the directory.

6.2 Exceptions

An agent reasoning in situations such as in the examples above, will undoubtedly infer false conclusions. As a result, some approach must be developed to identify the situations in which these conflicts occur, and then somehow resolve them. One possible solution is to simply remove a $K_w$ or LCW formula when an incomplete instance becomes known. However, this approach seems extreme since a single fact can cause the loss of all the universal assertions made by these formulas.

Instead, the approach adopted in this thesis is to identify the instances that do not apply to the $K_w$ and LCW formulas, and mark them as exceptions. For instance, in the first example, marking the file thesis.tex as an exception to the LCW formula would change the meaning of the LCW formula to indicate that the agent has local closed world information of the readability of all the files in the directory /project, except for the file thesis.tex.
6.2. EXCEPTIONS

6.2.1 Extending $K_w$ and $LCW$ Formulas to Model Exceptions

Exceptions make changes to the existing semantics of formulas in $K_w$ and $LCW$. As a result, a formula needs to be extended so that its representation takes into consideration the exceptions that are connected with it. To do this, every formula $\phi(\bar{x})$ in $K_w$ and $LCW$ is augmented to have associated with it the set $Except(\phi)$, indicating the bindings of free variables of $\bar{x}$ to which $\phi$ does not apply. For instance, the exception generated in the first example of this chapter would add the binding $x = \text{thesis.tex}$ to $Except(\phi)$.

Bindings in $Except(\phi)$ do not necessarily make use of all the free variables in $\bar{x}$. Partial bindings are permitted. For instance, if $\phi(x_1, x_2) = \alpha_1(x_1) \land \alpha_2(x_1, x_2)$ is in $LCW$ and the addition of the fact $\alpha_1(a)$ to $K_f$ generates an exception, then the binding $x_1 = a$ is added to $Except(\phi)$. When a formula $\phi(\bar{x})$ is initially added to $DB$, $Except(\phi)$ is set to $\emptyset$, indicating the exception listing is empty.

Formally, the set $Except(\phi)$ extends the definition of a formula in $K_w$ and $LCW$ as follows:

**Formulas in $K_w$:** For every formula $\phi(\bar{x}) \in K_w$ with exception set $E = Except(\phi)$, $KB$ includes the formula:

$$\forall \bar{x}. \bigwedge_{\bar{e} \in E} \neg(\bar{x} = \bar{e}) \Rightarrow K(\phi(\bar{x})) \lor K(\neg\phi(\bar{x})). \tag{6.1}$$

**Formulas in $LCW$:** Let $\phi(\bar{x}) \in LCW$ with exception set $E = Except(\phi)$. Let $C = \{\bar{c} | \alpha_i(\bar{x}/\bar{c}) \in K_f, 1 \leq i \leq k\}$. For every such formula $\phi(\bar{x}) \in LCW$ with exception set $Except(\phi)$, $KB$ includes the formula:

$$\forall \bar{x}. \bigwedge_{\bar{c} \in C \cup E} \neg(\bar{x} = \bar{c}) \Rightarrow K(\neg\phi(\bar{x})). \tag{6.2}$$

Exceptions are generated when the effects of an action cause formulas to be added or
removed from the \(K_f\) and \(K_v\) databases. In the next sections, the database updates that generate exceptions are discussed.

6.2.2 Exceptions Generated From Database \textit{add} Operations

An \textit{add} operation has the effect of adding a formula to a given database. Consider the case where a formula \(\varepsilon\) is added to \(K_f\) or \(K_v\), such that \(\varepsilon\) binds with one of the conjuncts of a \(K_w\) or \(LCW\) formula. Let \(\phi(\vec{x}) \in LCW\) (because of the parallels between \(K_w\) and \(LCW\) the same problem exists for the two. Without loss of generality, reference will be made to \(LCW\). However, the same process of reasoning applies to formulas in \(K_w\) as well). In the simplest situation, the formula \(\phi(\vec{x}) = \alpha_1(\vec{x})\) consists of a single conjunct. If \(\varepsilon = \alpha_1(\vec{x}/\bar{a})\) for some ground instance of \(\phi, \phi(\vec{x}/\bar{a})\), then no exception is generated. A complete instance of the formula \(\phi(\vec{x})\) (in this case \(\varepsilon\)) has been added to \(K_f\), so there is no conflict.

Now, consider the more general case where \(\phi(\vec{x}) = \alpha_1(\vec{x}) \land \ldots \land \alpha_k(\vec{x})\).\(^1\) The formula \(\varepsilon\) has the property that \(\varepsilon = \alpha_j(\vec{x}/\bar{a})\) for some \(j\) and some ground instance of \(\phi, \phi(\vec{x}/\bar{a})\). Before \(\varepsilon\) is added to \(K_f\) or \(K_v\), it is possible that no conjunct \(\alpha_i(\vec{x}/\bar{a})\) of \(\phi(\vec{x}/\bar{a})\) is known. A query such as \(IA(\alpha_i(\vec{x}/\bar{a}))\) would not return any information from step 6. Adding \(\varepsilon\) creates a situation in which there is a possibility that there is an incomplete instance of \(\phi\). That is, there may exist a conjunct \(\alpha_i(\vec{x}/\bar{a})\) for some \(i\), that is still not known. But, at the same time for the remaining conjuncts we have that \(IA(\alpha_j(\vec{x}/\bar{a})) = T\), for \(j \neq i\). Thus, reasoning about \(IA(\alpha_i(\vec{x}/\bar{a}))\) would return a value of \(F\) in step 6 of \(IA\). To avoid this incorrect inference, an exception is used. In particular, the binding of \(\varepsilon\) to \(\alpha_j(\vec{x}/\bar{a})\), \(\{\vec{x}/\bar{a}\}\) is added to the exception set.

It is also possible that adding \(\varepsilon\) may generate a complete instance of \(\phi\). However, to

\(^1\)If \(\varepsilon\) is being added to \(K_v\) and \(\alpha_j(\vec{x})\) is an equality of the form \(f(\vec{x}) = y\), then \(\varepsilon\) needs to be compared against \(f(\vec{x}/\bar{a})\), the left hand side of the equality.
6.2. EXCEPTIONS

avoid the computational expense required to determine this, a simpler, more conservative approach is taken, and an exception is always generated if a formula is added to $K_f$ or $K_v$ such that it binds with a conjunct of a $K_w$ or $LCW$ formula.

Adding a negative fact to $K_f$ does not result in any exceptions being generated. Recall that the conjuncts of formulas in $K_w$ and $LCW$ are restricted to being atomic. Consider a formula $\phi(\vec{x}) = \alpha_1(\vec{x}) \land \ldots \land \alpha_k(\vec{x}) \in LCW$ (the same reasoning applies for a formula in $K_w$), and a ground instance of $\phi$, $\phi(\vec{x}/\vec{a})$. The fact $\neg \epsilon$ is being added to $K_f$, where $\alpha_j(\vec{x}/\vec{a}) = \neg \epsilon$, for some $j$. If the inference algorithm is used to query $\alpha_i(\vec{x}/\vec{a})$, it will respond with $F$. If such a query is made to satisfy the conditions required in step 6 of $IA$ (for instance in condition (3) that $IA(\alpha_j(\vec{x}/\vec{a})) = T$), it will fail to do so. Thus, in this situation, the $LCW$ formula cannot be used to infer an incorrect result, so no exception is needed.

6.2.3 Exceptions Generated From Database delete Operations

A delete operation has the effect of removing a formula from a particular database. By doing so, an instance of a formula in $K_w$ or $LCW$ may become incomplete. Consider the removal of a formula $\epsilon$ from $K_f$ or $K_v$. Let $\phi(\vec{x}) = \alpha_1(\vec{x}) \land \ldots \land \alpha_k(\vec{x})$ be a formula in $LCW$ (again, the result is identical for a formula in $K_w$). The formula $\epsilon$ has the property that $\epsilon = \alpha_j(\vec{x}/\vec{a})$ for some $j$ and some ground instance of $\phi$, $\phi(\vec{x}/\vec{a})$. Before $\epsilon$ is deleted from $K_f$ or $K_v$, it is possible that every conjunct $\alpha_i(\vec{x}/\vec{a})$ of $\phi(\vec{x}/\vec{a})$ is known. A query such as $IA(\alpha_i(\vec{x}/\vec{a}))$ would return $T$. The act of removing $\alpha_j(\vec{x}/\vec{a})$ from the databases, has the effect of making this formula unknown. However, a query of $IA(\alpha_j(\vec{x}/\vec{a}))$ would return $F$ since $IA(\alpha_i(\vec{x}/\vec{a})) = T$ for the remaining conjuncts of $\phi$, satisfying the conditions in step 6 of $IA$. Thus, an exception is needed to prevent $IA$

---

2If $\epsilon$ is being deleted from $K_v$ and $\alpha_j(\vec{x})$ is an equality of the form $f(\vec{x}) = y$, then, again, $\epsilon$ needs to be compared against $f(\vec{x}/\vec{a})$, the left hand side of the equality.
from inferring the incorrect result. In this case, the binding of the conjunct $\alpha_j, \{\bar{x}/\bar{a}\}$, is added to the exception set.

A similar situation could occur if a negative fact is deleted from $K_f$. Whereas adding a negative fact does not cause a problem, deleting one could result in the same situation as above. Thus, the situation where $\neg \varepsilon = \alpha_j(\bar{x}/\bar{a})$ needs to be detected as well.

Finally, even if no formula is actually removed from a database by a delete operation, an exception may be generated. The effect of the delete operation makes a formula unknown. Thus, regardless of whether or not the formula is removed from a database, the effect of making it unknown must still be assessed.

Thus, if an delete operation is performed, such that the formula that is to be deleted binds with a conjunct of a $K_w$ or LCW formula, an exception is always generated.

### 6.2.4 Removal of Exceptions

The only situation that causes exceptions to be removed from a formula is the case where a formula $\phi$ is added to $K_w$ or LCW, but $\phi$ is already contained in the destination database, with $\text{Except}(\phi) \neq \emptyset$. In this case, $\text{Except}(\phi)$ is reset to $\emptyset$. In this situation the agent has obtained new local closed world information that overrides its existing LCW information. Thus, any exceptions that existed are now resolved (for the case of a formula in LCW, the list of facts satisfying it in $K_f$ is once again complete), so the exceptions are removed.

### 6.3 Exception Handling Rules

The rules for handling exceptions are summarized in Table 6.1. These rules implement the situations that were described above, and indicate the conditions that cause exceptions to be generated or removed. A very conservative approach is taken to exception
6.3. EXCEPTION HANDLING RULES

handling: “when in doubt, add an exception”. Thus, a disadvantage to this approach is that there are times when an exception may not be necessary, but is added by the rules. For instance, if in-dir(x, /project) ∈ LCW and readable(thesis.tex) ∈ K_f, and in-dir(thesis.tex, /project) is added to K_f, the exception x = thesis.tex is generated. An advantage to this approach is that the set of rules is kept simple and computationally efficient. The searching and reasoning required to determine if an instance of a K_w or LCW formula is completely satisfied, is avoided.

The exception rules are simply “wired” in with the primitive add and delete rules, so that when one of those procedures is invoked, the exception rules are executed as well. However, a difference between the two sets of rules is that the lack of a formula in a database (for instance on a delete operation such as in the second example in the chapter) may still cause an exception to be generated. Thus, the primitive database update rules are the trigger for the exception generation rules, but exception generation does not depend on the database operations making actual updates to the databases.

A small change needs to be made to the inference algorithm IA to allow it to work properly with the exception representation. In steps 6 and 7 of IA, condition (2) is changed to: α_i(\bar{x}/\bar{a}) = \varepsilon' for some i, such that the binding made to the conjunct \alpha_i, \{\bar{x}/\bar{a}\}, is not in Except(\phi).^3

For instance, step 6 would now be stated as:

6. If there exists a \phi(\bar{x}) = \alpha_1(\bar{x}) \land \ldots \land \alpha_k(\bar{x}) ∈ LCW and a ground instance of \phi, \phi(\bar{x}/\bar{a}), such that (1) \bar{a} are constants appearing in K_f, (2) \alpha_i(\bar{x}/\bar{a}) = \varepsilon' for some i, such that the binding made to the conjunct \alpha_i, \{\bar{x}/\bar{a}\}, is not in Except(\phi), and (3) IA(\alpha_j(\bar{x}/\bar{a})) = T for all j \neq i, then return (F).

This change indicates to the inference algorithm that it should not form a conclusion in

^3The binding made to \alpha_i may not necessarily involve all the variables in \bar{x}. Only the binding made to this particular conjunct is checked against the known exceptions.
1. For each \( \epsilon \) removed from \( K_f \) or \( K_w \), and for \( \phi(\vec{x}) = \alpha_1(\vec{x}) \land \ldots \land \alpha_k(\vec{x}) \) in \( K_w \) (LCW) such that:

(a) \( \alpha_i(\vec{x}/\vec{a}) = \epsilon \) for some \( i \) and substitution \( \vec{a} \), or

(b) \( \alpha_i(\vec{x}) \) is an equality of the form \( f(\vec{x}) = y \) and \( \epsilon \) is syntactically identical to \( f(\vec{x}/\vec{a}) \) (the left hand side of the equality), for some \( i \) and substitution \( \vec{a} \),

add the binding made to \( \alpha_i \{ \vec{x}/\vec{a} \} \), to \( \text{Except}(\phi) \).

2. For each \( \epsilon \) removed from \( K_f \), and for \( \phi(\vec{x}) = \alpha_1(\vec{x}) \land \ldots \land \alpha_k(\vec{x}) \) in \( K_w \) (LCW) such that \( \alpha_i(\vec{x}/\vec{a}) = \lnot \epsilon \) for some \( i \) and substitution \( \vec{a} \), add the binding made to \( \alpha_i \), \( \{ \vec{x}/\vec{a} \} \), to \( \text{Except}(\phi) \).

3. For each \( \epsilon \) added to \( K_f \) or \( K_w \), and for each \( \phi(\vec{x}) = \alpha_1(\vec{x}) \land \ldots \land \alpha_k(\vec{x}) \), \( k > 1 \), in \( K_w \) (LCW) such that:

(a) \( \alpha_i(\vec{x}/\vec{a}) = \epsilon \) for some \( i \) and substitution \( \vec{a} \), or

(b) \( \alpha_i(\vec{x}) \) is an equality of the form \( f(\vec{x}) = y \) and \( \epsilon \) is syntactically identical to \( f(\vec{x}/\vec{a}) \) (the left hand side of the equality), for some \( i \) and substitution \( \vec{a} \),

add the binding made to \( \alpha_i \), \( \{ \vec{x}/\vec{a} \} \), to \( \text{Except}(\phi) \).

4. If \( \phi(\vec{x}) = \alpha_1(\vec{x}) \land \ldots \land \alpha_k(\vec{x}) \) is added to \( K_w \) (LCW) and the formula \( \phi(\vec{x}) \) already exists in \( K_w \) (LCW) with \( \text{Except}(\phi) \neq \emptyset \), then set \( \text{Except}(\phi) = \emptyset \) (remove all exceptions).

Table 6.1: Exception handling rules

the case when the binding of \( \vec{a} \) to the free variables of \( \vec{x} \) in the conjunct \( \alpha_i \) is an exception. Thus, this change extends the handling of exceptions to the inference process.

The addition of exceptions has been presented as an extension to the existing knowledge representation framework, as opposed to being integrated with it as it was being developed in the earlier chapters. Exceptions are one approach to understanding and managing the interactions between different types of knowledge. This particular set of rules for handling exceptions leaves room for expansion, since it is very conservative in
its handling of exceptions. More work is needed to build a more sophisticated model of handling exceptions (Chapter 8 describes a possible extension), and the possibility remains that an alternate approach altogether may prove more feasible.

In the next chapter, examples of reasoning about sequences of actions are given, including an example that involves the generation of exceptions.
Chapter 7

Examples

In this chapter examples of action sequences are given, illustrating the use of the knowledge representation formalism. The examples highlight the changes made to an agent’s knowledge state during plan time, as the agent reasons about a sequence of actions, and during execution, as the sequence of actions is actually performed. The examples are taken from a variety of domains and demonstrate various features of the representation.

7.1 Open Safe Domain

The first example, due to Moore, is that of an agent trying to open a combination safe [Moo85]. There are two actions available to the agent in the domain, \textit{readComb} and \textit{dialComb}. Formal specifications for these actions are given in Table 7.1. The \textit{readComb} action allows an agent to read the combination for an object in the domain (for instance, the “safe” object), provided the agent actually has the combination. The \textit{dialComb} action allows the agent to try dialling a combination on the combination lock of a specified object. The agent’s initial knowledge state $S$ is described by the database $K_f = \{\text{haveComb}(\text{safe})\}$, indicating that the agent has the combination for the lock of the
safe object. The other databases are initially empty. Two different action sequences are considered for trying to achieve the goal of opening the safe.

The first action that the agent might consider is that of dialling a random combination on the safe, for example, the action `dialComb(safe, 27-54-13)`. Since `haveComb(safe)` is in $K_f$, then a query of $IA(haveComb(safe))$ will return a value of $T$ indicating that the first precondition, $K(haveComb(safe))$, is satisfied in $S$. As well, $K_v(27-54-13)$ is also satisfied. The inference procedure will return $W$ on the query $IA(27-54-13)$ since $27-54-13$ is a constant and all constants are known (step 2 of procedure $IA$). Thus, the agent knows at plan time that the preconditions of the action are satisfied.

The plan time effects of the action can now be applied to $S$. The `dialComb` action has a single conditional plan time effect. However, the condition $K(27-54-13 = combo(safe))$ cannot be inferred from $S$, since a query of $IA(27-54-13 = combo(safe))$ returns $U$. Thus, the conditional effect $add(K_f, open(safe))$ is not applied. Intuitively,
the agent does not know if the combination it is trying is the combination of the safe
(since in this instance it is simply trying a random combination) and thus does not know
if the action will cause the safe to open. Therefore, in this situation, no plan time effects
are applied to \( S \).

However, when the action is executed from the initial knowledge state \( S \), a different
set of effects will be applied. The combination 27-54-13 is passed to the execution mod-
ule along with the run time variable \(!safeopen\), as indicated by the \( \text{exec}(\text{dialComb}(x, y))\)
specification (where \( x \) is bound to \( \text{safe} \) and \( y \) is bound to 27-54-13). The execution mod-
ule will set \(!safeopen\) to \textbf{True} or \textbf{False} depending on whether or not the action succeeds
in opening the safe. If at execution time, \(!safeopen\) is set to \textbf{True} by the execution mod-
ule, then the action’s two conditional effects will be applied. The formulas \( \text{open}(\text{safe}) \)
and \( \text{combo}(\text{safe}) = 27-54-13 \) will be added to \( K_f \), updating the database, and creating a
new knowledge state, \( S' \). This means that if the safe opens then the agent comes to know
that it is open, and that the combination it tried is the right one. If, however, \(!safeopen\) is
set to \textbf{False}, no effects are activated and the knowledge state remains unchanged.\(^1\) Thus,
an agent may be lucky at execution time and the random combination may cause the safe
to open. However, at plan time, the agent is unable to conclude that the act of dialling
an arbitrary combination will cause the safe to open. Figure 7.1 illustrates the effects of
the \( \text{dialComb}(\text{safe}, 27-54-13) \) on the agent’s knowledge state, both at plan time and at
execution time.

Now consider the action \( \text{readComb}(\text{safe}) \) followed by \( \text{dialComb}(\text{safe}, \text{combo}(\text{safe})) \)
with the initial knowledge state given by \( S \). The precondition of the \( \text{readComb}(\text{safe}) \) ac-
tion, \( K(\text{haveComb}(\text{safe})) \), is satisfied in \( S \) so the effects of this action can be applied. At
plan time, the \( \text{readComb}(\text{safe}) \) action updates \( S \) by adding \( \text{combo}(\text{safe}) \) to \( K_v \), creating

\(^1\)Note that because restrictions are placed on the types of knowledge that can be described using the
formalism presented in this thesis, a formula such as \( \text{combo}(\text{safe}) \neq 27-54-13 \) cannot currently be repre-
sented.
Plan Time

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
<th>dialComb(safe, 27-54-13)</th>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>haveComb(safe)</td>
<td>$K_f$ haveComb(safe)</td>
<td>$K_f$</td>
<td>haveComb(safe)</td>
</tr>
</tbody>
</table>

Initial Knowledge State

Execution Time

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
<th>dialComb(safe, 27-54-13)</th>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>haveComb(safe)</td>
<td>$K_f$ haveComb(safe)</td>
<td>$K_f$</td>
<td>haveComb(safe)</td>
</tr>
</tbody>
</table>

Initial Knowledge State
!

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
<th>dialComb(safe, 27-54-13)</th>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>haveComb(safe)</td>
<td>$K_f$ haveComb(safe)</td>
<td>$K_f$</td>
<td>haveComb(safe)</td>
</tr>
</tbody>
</table>

Initial Knowledge State
!

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
<th>dialComb(safe, 27-54-13)</th>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>haveComb(safe)</td>
<td>$K_f$ haveComb(safe)</td>
<td>$K_f$</td>
<td>haveComb(safe)</td>
</tr>
</tbody>
</table>

Initial Knowledge State
!

Figure 7.1: Applying the $dialComb(safe, 27-54-13)$ action

a new knowledge state $S'$. Intuitively this means that the agent will come to know the combination of the safe (that is, the actual “value” of the combination).

Since $K(\text{haveComb}(\text{safe}))$ still holds in $S'$ (it was not deleted by the previous action), the first precondition of the $dialComb(\text{safe}, \text{combo}(\text{safe}))$ action is satisfied. Since $\text{combo}(\text{safe})$ was added to $K_v$ by the previous action, the second precondition holds in $S'$ as well. Thus, the plan time effects of this action can now be simulated on $S'$. To apply the single conditional plan time effect though, the condition $K(\text{combo}(\text{safe}) = \text{combo}(\text{safe}))$ must first be satisfied. Since the two terms of this condition are syntactically identical, the inference algorithm is able to return a value of $T$ on this query (step 3 of $IA$), even though $S'$ has nothing in it to allow $IA$ to simplify the terms. The condi-
7.1. OPEN SAFE DOMAIN

Plan Time

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
<th>readComb(safe)</th>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>haveComb(safe)</td>
<td>$K_f$ haveComb(safe)</td>
<td>$K_f$</td>
<td>combo(safe)</td>
</tr>
</tbody>
</table>

Initial Knowledge State

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
<th>dialComb(safe, combo(safe))</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>haveComb(safe)</td>
<td></td>
</tr>
<tr>
<td>$K_f$</td>
<td>open(safe)</td>
<td></td>
</tr>
<tr>
<td>$K_v$</td>
<td>combo(safe)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.2: Plan time effects of readComb–dialComb action sequence

The functional effect of the action is activated and the fact $open(safe)$ is added to the $K_f$ database of $S'$. This means intuitively, that at plan time the agent is able to reason that this action sequence will cause the safe to open, even though it doesn’t currently know what combination will actually be dialled. Figure 7.2 shows the plan time effects of these two actions on the agent’s knowledge.

At execution time, the $readComb(safe)$ action has the effect of determining what the value of the combination actually is. The execution module binds this value to the run time variable $val$, once it is retrieved from the environment. Suppose that this value is 15-42-7. The fact $combo(safe) = 15-42-7$ will be added to $K_f$ database in $S$. Also, the formula $combo(safe)$ is deleted from $K_v$.² The result is a new knowledge state, $S'$. Now the $dialComb(safe, combo(safe))$ action is executed in $S'$. Before any information is passed to the execution module, the terms are reduced using the $EvalTerm$ algorithm.

²This deletion is not strictly necessary, but has the effect of ‘cleaning up’ $K_v$. The primitive delete operation also performs this clean up if necessary, so as to maintain a consistent knowledge state.
CHAPTER 7. EXAMPLES

Execution Time

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>haveComb(safe)</td>
</tr>
<tr>
<td></td>
<td>!val = 15-42-7</td>
</tr>
</tbody>
</table>

Initial Knowledge State

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>haveComb(safe)</td>
</tr>
<tr>
<td>$K_f$</td>
<td>open(safe)</td>
</tr>
<tr>
<td>$K_f$</td>
<td>combo(safe) = 15-42-7</td>
</tr>
</tbody>
</table>

Figure 7.3: Execution time effects of $readComb$–$dialComb$ action sequence

The exec specification will be simplified to $exec(dialComb(safe, 15-42-7), !safeopen)$, with the second argument of $dialComb$ reduced to 15-42-7 by the function value added by the previous action. This reduction is important, and the reason why a $K_v(y)$ precondition is needed on the $dialComb$ action: the execution module cannot be expected to take as arguments, complex terms whose value is unknown. If the execution is successful, the execution module will bind $True$ to the run time variable, $!safeopen$, causing $open(safe)$ to be added to $K_f$ in $S'$. Since, $combo(safe) = 15-42-7$ is already in $K_f$, the second $add$ operation does nothing. The execution time effects of this action sequence on the agent’s knowledge are shown in Figure 7.3.

7.2 Medical Domain

The second example describes a simplified medical domain and is due to Weld, Anderson, and Smith [WAS98]. Three actions are available in the domain: $drink$, $medicate$, $"
### 7.2. MEDICAL DOMAIN

<table>
<thead>
<tr>
<th>Command</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>drink</strong></td>
<td><strong>Plan Time:</strong> &lt;br&gt;add($K_f$, hydrated) &lt;br&gt;<strong>Execution Time:</strong> &lt;br&gt;add($K_f$, hydrated)</td>
</tr>
<tr>
<td><strong>medicate</strong></td>
<td><strong>Plan Time:</strong> &lt;br&gt;$K$(hydrated) $\Rightarrow$ add($K_f$, $\neg$infected) &lt;br&gt;$K$(hydrated) $\Rightarrow$ add($K_f$, dead) &lt;br&gt;$\neg K_w$(hydrated) $\Rightarrow$ delete($K_f$, $\neg$dead) &lt;br&gt;<strong>Execution Time:</strong> &lt;br&gt;exec(medicate, $!$alive) &lt;br&gt;!$alive = False $\Rightarrow$ add($K_f$, dead) &lt;br&gt;!$alive = True $\Rightarrow$ add($K_f$, $\neg$infected)</td>
</tr>
<tr>
<td><strong>stain</strong></td>
<td><strong>Plan Time:</strong> &lt;br&gt;add($K_w$, blue), add($K_w$, infected) &lt;br&gt;<strong>Execution Time:</strong> &lt;br&gt;exec(stain, $!$stainblue) &lt;br&gt;delete($K_w$, blue), delete($K_w$, infected) &lt;br&gt;!$stainblue = True $\Rightarrow$ add($K_f$, blue), add($K_f$, infected) &lt;br&gt;!$stainblue = False $\Rightarrow$ add($K_f$, $\neg$blue), add($K_f$, $\neg$infected)</td>
</tr>
</tbody>
</table>

Table 7.2: Medical domain actions

and stain. Formal descriptions of these actions are given in Table 7.2. The goal is to cure a patient’s infection, if the patient has one, without killing the patient. However, it is not initially known whether or not the patient is infected. The drink action has the effect of hydrating the patient. medicate has the ability to cure the infection, but only if the patient is hydrated. Otherwise, it will kill the patient. The stain action can be used to test if the patient is infected: the stain becomes blue if the patient is infected. None of these actions have preconditions, so only the effects need to be considered. The agent's initial knowledge state is given by $K_f = \{\neg$dead $\}$, with the other databases empty.

One possible plan is the action sequence drink followed by medicate. At plan time, drink has the effect that the agent knows that the patient is hydrated. The second action,
medicate, has three conditional plan time effects. Since the agent knows hydrated, the first conditional effect is activated and the agent will also come to know $\neg$infected. The second conditional effect is not applied, and since $K(\text{hydrated})$ implies $K_w(\text{hydrated})$, the third conditional will not be activated either. Neither of these actions had effects that caused $\neg$dead to be removed from $K_f$ so the agent can conclude that the patient will be alive after applying this action sequence. Thus, the agent is able to construct a plan that it knows at plan time will achieve the goal of curing the patient.

A second possible plan is the single action medicate, performed without first hydrating. Since the agent does not initially have any knowledge about hydration, the third conditional effect is activated. The fact $\neg$dead is removed from $K_f$ and the agent loses its knowledge that the patient is not dead. Therefore, at plan time the agent can conclude that this plan is not safe since the medicate action has the effect of making dead unknown.

Now consider the plan described by the action stain, followed by the conditional action sequence: if $K(\text{infected})$ then drink followed by medicate. At plan time, the stain action has the effect of adding infected to $K_w$. Thus, the agent knows that after executing the stain action its knowledge state will either contain the fact infected or $\neg$infected. Given any ground formula $\phi$ (or an instance of a non-ground formula) in $K_w$, the planner is able to add conditional branches into the plan, assuming along one branch that $\phi$ is known to the agent ($\phi$ is added to $K_f$), while along the other branch that $\neg\phi$ is known ($\neg\phi$ is added to $K_f$). Plan generation can then proceed along each plan branch as normal. At execution time the $K_w$ formula will be resolved and the plan executor will be able to determine which branch of the plan to take.

At this point in the plan then, following the stain action, two conditional branches can be added to the plan, since infected is in $K_w$. Along one of the branches, the planner assumes infected to be true and adds infected to $K_f$. Along the other branch, it assumes
7.2. MEDICAL DOMAIN

Plan Time

Initial Knowledge State

Figure 7.4: Plan time effects of a conditional action sequence in the medical domain

infected to be false and adds \( \neg \text{infected} \) to \( K_f \). Planning proceeds to complete the plan along each of the plan branches, ensuring that the goal is reached in each contingency.

One branch of the plan starts in a state where \( K_f = \{ \neg \text{dead}, \text{infected} \} \). Now, by applying the drink and medicate actions, the agent’s knowledge is changed to a state where it knows the patient is \( \neg \text{infected} \). The other branch of the plan starts in a state where \( K_f = \{ \neg \text{dead}, \neg \text{infected} \} \). In this case, no additional actions are needed since the goal has been achieved. Thus, the agent is able to reason at plan time that the conditional plan achieves its goal. Figure 7.4 illustrates the plan time effects of the conditional plan on the agent’s knowledge.

At execution time, when the stain action is executed, the execution module is able to determine if the colour of the stain is blue or not, and binds this result to the run time variable \( \text{stainblue} \). The truth value of this variable is then used to determine which conditional effect should be applied. The result is that either infected or \( \neg \text{infected} \) is
added to $K_f$. In either case, the plan executor will have enough information to correctly execute the rest of the conditional plan.

Thus, this conditional plan satisfies Levesque’s requirement that at plan time the agent must know the plan will achieve the desired effect, and that at execution time, the agent must have sufficient knowledge to execute the plan [Lev96]. By considering the possible consequences of the first action, the agent is able to plan appropriately and guarantee at plan time, that the goal of curing the patient’s infection will be achieved. However, it only becomes known at execution time, which branch of the plan will be executed to achieve the goal (either drink and medicate, or doing nothing).

7.3 UNIX Domain: Filesystem Actions

The third example is taken from the UNIX domain. Two UNIX actions are available to the agent, ls and compress, with their formal specifications given in Table 7.3.³ An agent in this domain tries to satisfy the goal “if file thesis.tex is in directory /project and readable, then compress it”, by considering the following conditional plan: ls -al /project, if $K(\text{in-dir(thesis.tex, /project)})$ and $K(\text{readable(thesis.tex)})$ then compress thesis.tex. The agent’s initial knowledge state $S$ is given by the database $K_f = \{\text{readable(/project), readable(thesis.tex)}\}$, with the other databases empty.

Using the action specifications, the conditional plan can be projected to determine the agent’s knowledge state at plan time. The preconditions of the ls -al /project action hold in the initial state, so the plan time effects can be applied to $S$. This generates a new knowledge state, $S'$, where the $K_f$ database remains unchanged, but $K_w = \{\text{in-dir}(x, /project), \text{in-dir}(x, /project) \land \text{readable}(x), \text{in-dir}(x, /project) \land \text{size}(x) = y\}$. Since $\text{in-dir}(x, /project)$ is added to $K_w$ then $S'$ entails $K_w(\text{in-dir(thesis.tex, /project)})$

³The specifications for the UNIX actions have been simplified somewhat.
Table 7.3: UNIX domain filesystem actions

<table>
<thead>
<tr>
<th>Command</th>
<th>Precondition</th>
<th>Effects</th>
</tr>
</thead>
</table>
| `ls -al z` | $K(\text{readable}(z))$ | **Plan Time:**
|           |                | $\text{add}(K_w, \text{in-dir}(x, z))$
|           |                | $\text{add}(K_w, \text{in-dir}(x, z) \land \text{readable}(x))$
|           |                | $\text{add}(K_w, \text{in-dir}(x, z) \land \text{size}(x) = y)$
|           |                | **Execution Time:**
|           |                | $\text{exec}(\text{ls -al z}; !\text{file}; !\text{readable}; !\text{size})$
|           |                | $\text{add}(K_f, \text{in-dir}(!\text{file}; z))$
|           |                | $!\text{readable} = \text{True} \Rightarrow$
|           |                | $\text{add}(K_f, \text{readable}(!\text{file}))$
|           |                | $\text{add}(K_f, \text{size}(!\text{file}) = !\text{size})$
|           |                | $\text{add}(LCW, \text{in-dir}(x, z))$
|           |                | $\text{add}(LCW, \text{in-dir}(x, z) \land \text{readable}(x))$
|           |                | $\text{add}(LCW, \text{in-dir}(x, z) \land \text{size}(x) = y)$
| \text{compress} $x$ | $K(\text{readable}(x))$ | **Plan Time:**
|           |                | $\text{delete}(K_v, \text{size}(x))$
|           |                | **Execution Time:**
|           |                | $\text{exec}(\text{compress } x)$
|           |                | $\text{delete}(K_f, \text{size}(x))$
|           |                | $\text{delete}(K_v, \text{size}(x))$

(this is an instance of the $K_w$ formula). The planner can now add two conditional branches into the plan, based on the $K_w$ formula $\text{in-dir}(\text{thesis.tex}, /\text{project})$. Thus, the branch in the second step of the conditional plan is legitimate.

Along the false branch of the plan, the agent knows $\neg\text{in-dir}(\text{thesis.tex}, /\text{project})$. This is sufficient to conclude that the goal has been achieved along this branch. Along the other branch of the plan, the agent knows $\text{in-dir}(\text{thesis.tex}, /\text{project})$. Also, the agent knows $\text{readable}(\text{thesis.tex})$ since this formula is still in $K_f$ (the $ls$ action did not remove it). Thus, the preconditions for applying the \text{compress} \text{thesis.tex} action in this branch are satisfied.
Applying this action activates the plan time effect of deleting $\text{size}(\text{thesis.tex})$ from $K_v$. Even though this formula is not explicitly in $K_v$, the delete operation causes an exception to be generated for the $K_w$ formula $\text{in-dir}(x, /\text{project}) \land \text{size}(x) = y$. Removing the size of $\text{thesis.tex}$ would cause an instance of this formula to be incomplete and as a result, the exception handling rules generate the exception $x = \text{thesis.tex}$ and add it to the exception set for $\text{in-dir}(x, /\text{project}) \land \text{size}(x) = y$. This indicates that the closed world formula no longer applies when $x$ is $\text{thesis.tex}$. The goal has now been achieved along this branch of the plan. Figure 7.5 shows the plan time effects of this conditional plan.

Consider the conditional action sequence at execution time. Assume that in the real world we have $\text{in-dir}(\text{thesis.tex}, /\text{project})$, $\text{readable}(\text{thesis.tex})$, and $\text{size}(\text{thesis.tex}) = 50000$. Executing the $\text{ls -al /project}$ action has the effect of adding this information to the agent’s $K_f$ database. The formulas $\text{in-dir}(x, /\text{project})$, $\text{in-dir}(x, /\text{project}) \land \text{readable}(x)$, and $\text{in-dir}(x, /\text{project}) \land \text{size}(x) = y$ are also added to $LCW$. Since the agent knows $\text{in-dir}(\text{thesis.tex}, /\text{project})$ and $\text{readable}(\text{thesis.tex})$, the “true” branch of the conditional plan is followed. The preconditions for the $\text{compress \text{thesis.tex}}$ action are satisfied and applied, causing the formula $\text{size(thesis.tex)} = 50000$ to be deleted from $K_f$. This database update also generates the exception $x = \text{thesis.tex}$ to be added to the exception set of the $LCW$ formula $\text{in-dir}(x, \text{thesis.tex}) \land \text{size}(x) = y$, indicating that the agent knows the size of all files in /project, except for $\text{thesis.tex}$.

Now consider another possible configuration of the real world, in this case the situation where $\text{in-dir}(\text{thesis.tex}, /\text{papers})$. After the $\text{ls -al /project}$ action the agent does not know that $\text{in-dir}(\text{thesis.tex}, /\text{project})$. But, the $LCW$ formula $\text{in-dir}(x, /\text{project})$ lets the agent reason that it knows $\neg \text{in-dir}(\text{thesis.tex}, /\text{project})$. In this case, the plan executor still has enough knowledge to decide which branch of the conditional plan to take. The “false” branch is followed, no further actions are taken, and the goal is satisfied. Thus,
7.3. **UNIX DOMAIN: FILESYSTEM ACTIONS**

<table>
<thead>
<tr>
<th>Plan Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Knowledge State</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>readable(/project)</td>
</tr>
<tr>
<td>$K_f$</td>
<td>readable(thesis.tex)</td>
</tr>
</tbody>
</table>

$ls$ -al /project

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>readable(/project)</td>
</tr>
<tr>
<td>$K_f$</td>
<td>readable(thesis.tex)</td>
</tr>
<tr>
<td>$K_w$</td>
<td>indir(x, /project)</td>
</tr>
<tr>
<td>$K_w$</td>
<td>indir(x, /project) ∧ readable(x)</td>
</tr>
<tr>
<td>$K_w$</td>
<td>indir(x, /project) ∧ size(x) = y</td>
</tr>
</tbody>
</table>

~ indir(thesis.tex, /project) \[\rightarrow\] indir(thesis.tex, /project)

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>readable(/project)</td>
</tr>
<tr>
<td>$K_f$</td>
<td>readable(thesis.tex)</td>
</tr>
<tr>
<td>$K_f$</td>
<td>indir(thesis.tex, /project)</td>
</tr>
<tr>
<td>[\vdots]</td>
<td>[\vdots]</td>
</tr>
</tbody>
</table>

compress thesis.tex

<table>
<thead>
<tr>
<th>db</th>
<th>entry</th>
<th>exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_f$</td>
<td>readable(/project)</td>
<td></td>
</tr>
<tr>
<td>$K_f$</td>
<td>readable(thesis.tex)</td>
<td></td>
</tr>
<tr>
<td>$K_f$</td>
<td>indir(thesis.tex, /project)</td>
<td></td>
</tr>
<tr>
<td>[\vdots]</td>
<td>[\vdots]</td>
<td></td>
</tr>
<tr>
<td>$K_w$</td>
<td>indir(x, /project) ∧ size(x) = y</td>
<td>[x = \text{thesis.tex}]</td>
</tr>
</tbody>
</table>

Figure 7.5: Plan time effects of a conditional action sequence in the UNIX domain
at plan time the agent is able to reason that it knows the conditional plan will achieve its goals along each branch. At execution time, the agent gains sufficient knowledge from the environment to decide which branch of the plan should actually be executed.

7.4 UNIX Domain: Network Actions

This example is also taken from the UNIX domain, however it utilizes the UNIX commands `mail`, `finger`, and `ping`. The formal specifications for these actions are given in Table 7.4.

The `mail` action has the ability to deliver a file `z` to a user `x` on machine `y`. However, the only guarantee of delivery is if the agent knows the user is on the machine and that the machine is active. Otherwise, the delivery status is unknown (the message could potentially be “lost” on the network, or sitting in the message queue if the destination machine is down). The `finger` action can be used to determine if a user `x` has an account on machine `y`, but only if `y` is active. The `ping` action can be used for querying the status of machine `y`.

First, consider an agent trying to achieve the goal of “send the file `letter` to user `rpapetri`”. The agent’s initial knowledge state is described by $K_f = \{\text{readable}(\text{letter}), \text{has-account}(\text{rpapetri}, \text{logos}), \text{has-account}(\text{rpapetri}, \text{neumann})\}$, and the database $LCW = \{\text{has-account}(\text{rpapetri}, x)\}$. The other databases are empty. One plan an agent may try is simply `mail rpapetri logos < letter`. At plan time the agent is unable to determine whether or not the file will get delivered, since it is not known if `active(logos)`. Thus, the conditional plan time effect will not get activated. Also, actually executing this command does not produce an effect that is known to the agent, again, since the agent knows nothing about the status of the machine `logos`. The message may or may not get delivered, but the agent will not know either way.

\footnote{Again, these actions have been simplified.}
Another possible plan may be the action sequence `mail rpapetri logos < letter`, followed by `mail rpapetri neumann < letter`. As with the previous plan, the delivery status of `letter` will not be known to the agent at plan time or at execution time. However, since the agent has `LCW` information about the machines that `rpapetri` has accounts on, this plan has the effect of delivering the message to user `rpapetri` if possible. The agent could reason that in this case it is covering all situations by simply mailing the file to all the addresses it knows could satisfy the request. But, with the delivery status unknown, this plan sequence could have the effect of delivering the file `letter` to both accounts, which may not be the desired effect that the user who posted the goal to the agent had in mind.
Another goal to consider would be the goal “if user rpapetri is on logos and logos is active then deliver the file letter to that account, otherwise, if user rpapetri is on neumann and neumann is active then deliver the file letter to that account”. Consider the initial knowledge state $K_f = \{\text{readable(letter)}\}$, with the other databases empty. In this case, the conditional plan ping logos, if $K(\text{active(logos)})$ then finger rpapetri, if $K(\text{has-account(rpapetri, logos)})$ then mail rpapetri logos < letter could be used to satisfy the first part of the goal.

At plan time, the ping action determines $K_w(\text{active(logos)})$. Using this knowledge the planner can build a conditional branch for the case when the agent knows active(logos). The preconditions of finger rpapetri are satisfied on this branch so the agent will come to know $K_w(\text{has-account(rpapetri, logos)})$. Again, a conditional branch can be added for the case when the agent knows has-account(rpapetri, logos). Now, when reasoning about the effects of mail, the plan time effect is activated and the agent is able to conclude delivered(letter, rpapetri). Each of the “false” branches in the above plan can be used as the starting point to try to satisfy the second part of the goal condition. In this case, a similar action sequence as the one used for logos can be used for neumann. At execution time, the plan executor will be able to resolve which branch to take in each case. The actual result of the plan (delivering the file letter to logos, neumann, or not delivering it at all) will only be known at execution.
Chapter 8

Future Work and Conclusions

The next stage of the research is discussed in this chapter, along with some of the short term and long term goals of the work. Possible extensions are described indicating directions that future research can take. The work presented in this thesis is summarized and the contributions and conclusions it makes are discussed.

8.1 Implementation

The research presented in this thesis has been primarily theoretical. The next step will be to implement a prototype of the planner, using the algorithms and constructs described here. An existing planner, such as TLPAN [BK96b], can be used as a starting point and modified to support the database framework that has been developed for modelling an agent’s incomplete knowledge. The inference algorithm, update rules, and exception handling rules can be implemented reasonably easily. The existing TLPAN action representation framework also has to be adapted to allow it to interact with the new collection of databases.

An execution module can also be implemented. This would require constructing an
CHAPTER 8. FUTURE WORK AND CONCLUSIONS

interface to interact with the environment, and communicate information back to the representation module in a useful form. For instance, in the UNIX domain, a module could be built to interact with the UNIX environment by using a modified UNIX shell. This shell would execute the specific UNIX commands required by the action specifications, gather the information returned from such commands, and pass this information back to the execution module. The data can then be parsed and bound to the appropriate run-time variables which in turn are passed to an interpreter that applies the execution-time effects of the action. Additional bindings of data to run-time variables can be requested by the interpreter.

Once implemented, it would be possible to test the prototype in sample domains. In particular, an analysis of the prototype’s performance will help to assess the value of the approach, and allow us to compare it against that of other existing planners. Testing with the prototype will also help provide a better understanding of some of the problems that may be faced in practice, for instance in domains such as the UNIX domain. This will no doubt lead to refinements in the approach, and future research.

By using an existing planner such as TLPLAN we can inherit its forward chaining approach to planning. Given the projective action semantics of the research presented here, this is a reasonable approach to take. In order to support partial order planning or backwards chaining planning more work would have to be done to adapt this approach to provide the kinds of information such planners require.

The main goal of such a prototype would be to show that this representation can be used effectively in practical planning systems. The lessons learned from testing the prototype will help in the construction of a more efficient and functional planner. Such a tool can then be used as a component in building more complex systems. In particular, the planner can then be used to help achieve the long term goal of constructing a usable software agent, such as an operating system or Internet agent.
8.2 Extensions

One possible extension of the current work could involve modifying the exception handling rules. Currently, the rules are very conservative in their updates of $K_w$ and $LCW$ formulas. Exceptions are added, but they are never removed unless an action has the effect of “adding” the same formula again to $K_w$ or $LCW$, causing all exceptions to be removed from that formula. It may be possible to expand the current approach to allow exceptions to be both added and removed, depending on the effects of actions. For instance, if an exception is caused by deleting facts from $K_f$, it may be possible to remove the exception by adding the facts that were deleted, back into $K_f$. However, the effectiveness of the current approach needs to be weighed against the changes a new procedure would make to the existing representation and update procedures. Testing with an implemented prototype will help to determine the worth of an exception handling procedure in practice.

Another possible extension is to expand the types of knowledge the planner is able to represent. In particular, knowledge of a subset of universally quantified formulas, such as formulas of the form $\forall x. \phi(x) \Rightarrow \psi(x)$, may be useful to a planning agent. For instance an agent may know the universal formula that $\forall x. \text{postscript}(x) \Rightarrow \text{printable}(x)$. This captures the idea that “all postscript files are printable”. Such knowledge may help an agent build more effective plans, by allowing it to make more powerful inferences. However, much additional work would be needed to properly understand and utilize it.

A second useful type of knowledge that could be added to the representation is indexical knowledge. Lesperance and Levesque argue that many times the knowledge required for action is generally relative or indexical rather than absolute or objective, especially when dealing with incomplete knowledge. An agent may have sufficient knowledge to achieve its goals even if it doesn’t know its identity, its location, the time, what objects
are around it, or the location (in absolute terms) of these objects [LL95]. For instance, a robot that knows that a tool is located at coordinates (10, 42, 23) may still not be able to get to the tool if it doesn’t know its own location, or how to find its way to the tool’s coordinates. On the other hand, if the robot has indexical knowledge that the tool is “three metres to its left”, it is able to go and pick up the tool without needing to know its own absolute position or the absolute position of the tool. Such knowledge would also be useful in domains such as the UNIX domain, where humans often use indexical knowledge to achieve goals such as “make the parent directory readable”, even though the user may not know the name or location of the parent directory. More work would be needed though to overcome the problem of representing indexical knowledge in a manner that is useful to the agent.

An important extension to this research would be to address the problem of “harmful” actions. Without some additional information to guide the planner to a “safe” plan, there is an inherent danger that the actions of the plan may be harmful if blind search is used to bring about the goal state. For instance, an operating system agent trying to preserve disk space by simply deleting files in the filesystem (instead of say, first trying to just compress the files) can cause harm if valuable files are destroyed. Weld and Etzioni explore how to formalize the notion of harm and to avoid performing harmful actions, by using annotations that act as safety constraints when specifying goal conditions [WE94]. However, the types of conditions that can be represented is very restrictive. Bacchus and Kabanza provide a more expressive approach to addressing the problem of safety and maintenance goals, by using temporal constructs [BK96a]. Their temporal language allows a richer collection of goal conditions to be expressed, compared with the use of annotations. Moreover, their language could be adapted to work with the framework described in this thesis. Given that the semantics of this representation are inherently projective, formalizing temporal operators that place conditions on knowledge
as it moves through future states seems a natural extension.

Another possible extension would take this research in a different direction, by allowing the planner to execute multiple actions concurrently. This ability allows us to model an agent that has multiple sensors and effectors, or to model a “restricted” form of *multi-agent system* (MAS) in which the agents are simple program-execution modules under the control of a central planner (each module can be thought of as executing part of a larger plan, returning all information gathered from the environment to the central planner). Boutilier and Brafman describe a way to extend the STRIPS action representation language to represent concurrent interacting actions [BB97]. Their approach adds a *concurrent action list* to the STRIPS representation that describes the restrictions on the actions that can or cannot be executed concurrently, for the action to have its specified effect. The action representation presented in this thesis could be similarly adapted for concurrent actions.

### 8.3 Contributions and Conclusions

This thesis describes a formalism for modelling an agent’s incomplete knowledge. The standard STRIPS representation is extended to allow an agent’s knowledge to be represented by a collection of databases, with each database storing a specific type of knowledge. The agent’s knowledge state is formally defined by providing a translation of the database contents to formulas in modal logic. Thus, the logic’s semantics are used as the underlying semantics of the representation. The agent’s knowledge can include knowledge of literals, functions, know whether formulas, and local closed world information.

A sound but incomplete inference procedure is described that allows queries of the agent’s knowledge to be made. A primitive query language is developed that relates knowledge conditions to the results of queries made with the inference algorithm. Sat-
isfying a knowledge condition about some formula reduces to an inference algorithm query returning the correct result about a corresponding formula.

A STRIPS-like approach to representing an agent’s actions is also developed. An action’s preconditions are specified as a list of primitive queries. An action’s effects are divided into its plan time and execution time effects. Each set of effects is described by a conditional list of add and delete operations that allow updates to be made to specific databases. Algorithms for the primitive database \textit{add} and \textit{delete} operations are given. These algorithms preserve the conditions necessary for maintaining a consistent knowledge state. The notion of an exception is developed as a means of managing the interaction between formulas in $K_f$ and $K_v$ and formulas in $K_w$ and $LCW$. Examples are given that illustrate reasoning about sequences of actions both at plan time and at execution time. The examples also demonstrate how the knowledge representation, action representation, inference procedure, and database update rules interact.

This thesis makes a number of contributions. First, the knowledge representation framework that is developed provides a structure for modelling the effects of knowledge-producing actions and actions that produce closed world effects. The knowledge representation formalism focuses on the types of knowledge an agent can represent. For instance, the $K_w$ and $K_v$ databases can be used to model the plan time effects of sensing actions, such as those that sense the truth of a formula, or the value of a function. More complex formulas in $K_w$ and $K_v$ can be used to model the plan time effects of actions with universal closed world effects. The $LCW$ database provides the means of representing the execution time effects of actions with closed world effects. Since actions are described by their effects on the agent’s knowledge state, actions with effects that both sense and manipulate the environment can be represented without requiring special attention. Moreover, the representation is given both formally and algorithmically, describing a mechanism for representing, updating, and querying the knowledge state of
the agent. The exception handling rules provide a means of preserving local closed world information in situations where such knowledge interacts with other types of knowledge.

Second, a clear separation between the plan time and execution time effects of an action is given. The effects of an action at plan time are often quite different from the effects of an action at execution time. Being able to project an agent’s knowledge through the plan time effects of a sequence of actions gives the agent the ability to construct plans that it knows (at plan time) will achieve its goals. Moreover, a simple forward chaining approach could be used in practice to do this. The agent’s knowledge state can also be projected through the execution time effects of an action sequence. The ability to do so at execution time means that an execution module can be constructed to support this formalism. Separating an action’s plan time effects from its execution time effects can help us better understand the tradeoffs between planning and execution, and the issues concerning the interleaving of the two.

Third, the formalism also supports contingency planning. The $K_w$ and $K_v$ databases both represent types of knowledge that an agent will come to know. As a result, a planner can introduce conditional plan branches that are based on the possible resolutions of formulas in $K_w$ and $K_v$. For instance, if a formula is in $K_w$, a plan branch could be generated based on the assumption that the agent knows the formula, while a second branch could be generated with the assumption that the agent knows the negation of the formula. If a formula is in $K_v$, plan branches can be constructed based on the agent knowing particular values of the $K_v$ formula.

Finally, this thesis tries to provide a representation that is formally characterized, yet still effective for building practical systems. The knowledge representation formalism is presented with a rigorous description of the types of knowledge that can be represented, how that knowledge can be updated, and how the effects of actions can be used to project an agent’s knowledge state. However, the formalism was also developed with
a focus towards the actual construction of useful planning systems. The algorithms that are developed keep in mind issues of computational complexity and efficiency. The examples that are given provide preliminary evidence that the representation can be used effectively in practice.

An implementation, however, is necessary to illustrate the effectiveness of the knowledge representation formalism in practice. Testing in a variety of domains will help to establish the strengths and limitations of the approach. Moreover, an implementation is essential for assessing how certain algorithms, such as the inference procedure and the exception handling rules, actually perform in domains such as the UNIX domain. It is also hoped that insights can be gained into some of the important issues yet to be solved, such as the tradeoffs between planning and execution, and the best way to handle the interactions between knowledge of different types. The prospect of constructing a prototype is also important since such a module can then be used as the basis for building more complex applications, such as a usable software agent.
Bibliography


[EGW94] Oren Etzioni, Keith Golden, and Daniel Weld. Tractable closed world rea-


