The Dissertation Committee for Selim Turhan Erdoğan
certifies that this is the approved version of the following dissertation:

A Library of General-Purpose Action Descriptions

Committee:

Vladimir Lifschitz, Supervisor

Michael Gelfond

Benjamin J. Kuipers

Bruce Porter

Peter Stone

A Library of General-Purpose Action Descriptions

by

Selim Turhan Erdoğan, B.S.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2008

To my family

Acknowledgements

I would like to thank my advisor Vladimir Lifschitz for making this dissertation possible. This project grew from the seed of an idea he recommended initially, and my work thereafter has been guided by his many incisive observations and helpful suggestions. His very precise analysis of all problems he encounters and his formal approach always lead to great insight into the research questions we encounter. These are skills that I hope to have at least partially acquired during my time working with him and I feel grateful to have been lucky enough to be his student.

I also wish to thank the other members of my dissertation committee, for serving on my committee and for their useful comments on my dissertation. Michael Gelfond is a master of attention to detail and helped me clarify several aspects of this work. Ben Kuipers encouraged me to think about the relation of my research to researchers in other areas. Bruce Porter shared his experience in building large knowledge bases and helped me keep the long-term goals of such projects in mind. Peter Stone's comments helped me think about how the success of such work may be measured. I would also like to acknowledge his friendly guidance and leadership during the time I spent working with the Austin Villa robot soccer team at U.T.

I have had the privilege to be in a research group with many wonderful graduate students and I extend my thanks to Esra Erdem, Paolo Ferraris, Joohyung Lee, Yuliya Lierler, and Wanwan Ren for all of our good times during the interesting conversations in the office, the trips to conferences, and the parties.

Studying at a great school like U.T. gave me the opportunity to meet many smart and nice people. Thanks to all my friends in the Computer Sciences department and also those from the Turkish University Students Association. They are too numerous to mention. However, I would especially like to thank Özgür Erciyes, for his exceptional friendship, support and encouragement.

The National Science Foundation provided partial financial support for my research.

The greatest support for everything I have done in my life has come from my parents, Turhan and Phyllis Erdoğan, and my brother, Sinan Erdoğan. My parents were always by my side, even with the many miles between us, constantly encouraging me, keeping my spirits up, and sharing their experience. Sinan was not only my brother, but my roommate, best friend, and most ardent supporter in Austin. Thanks to them from the bottom of my heart.

SELİM TURHAN ERDOĞAN

The University of Texas at Austin

August 2008

A Library of General-Purpose Action Descriptions

Publication No. _____

Selim Turhan Erdoğan, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Vladimir Lifschitz

An important idea in knowledge representation is that of libraries of reusable knowledge components. The goal of this research is to apply this idea to action description languages. An action language may be used to specify the effects and preconditions of actions, and serves to describe "transition systems" — directed graphs with the vertices representing the states of an action domain and the edges representing the transitions that are caused by performing actions (or by the passage of time). Many actions can be described as special cases of other actions. (For example, pushing, carrying, going can all be described as special cases of moving things around.) However, descriptions of action domains in existing action languages describe the effects of all actions from scratch, which leads to common aspects of different domains getting reinvented over and over.

In this dissertation we first developed a method for defining actions in terms of other actions, in the action language $\mathcal{C}+$, a language with a rich set of features for describing action domains. This provided a theoretical basis for developing a library of general-purpose action descriptions and influenced the design of the Modular Action Description (MAD) language [Lifschitz and Ren, 2006], the semantics of which is based on $\mathcal{C}+$.

We extended the original MAD language in several ways, both in the syntactic dimension and in the semantic dimension, and developed an implementation of this extended language. The extended semantics not only provides new features but also addresses some shortcomings of the original semantics, which were identified during the course of our research. The implemented system was used to develop a library of basic MAD modules, each describing a group of general commonsense facts related to actions.

Several action domains from the knowledge representation literature were formalized using the library of basic action descriptions. The availability of the library led to the representations being much simpler than before and also enabled us to recognize structural similarities of seemingly quite different domains.

Contents

Chapter 1

Introduction

Two important lines of research in artificial intelligence are reasoning about actions and the construction of general-purpose knowledge bases. We would like to bring these two areas closer together.

The area of reasoning about actions has seen a great deal of progress in the last twenty years. The tradition has been to try to formalize small examples very precisely and see where things go wrong. This leads to the invention of new formalisms that are more and more expressive, capable of capturing information that wasn't possible to represent before. This approach has resulted in the identification and solving of many important problems. In particular, the frame problem [McCarthy, 1979] and the ramification problem [Finger, 1986] have been solved using nonmonotonic formalisms [Shanahan, 1997, Geffner, 1990, Lin, 1995, McCain and Turner, 1997]. There are several implemented systems with very expressive input languages, and the ability to solve action problems such as planning or prediction. These systems have been used to formalize many small- and medium-sized domains [Lifschitz *et al.*, 2000, Lifschitz, 2000, Campbell and Lifschitz, 2003,

Akman *et al.*, 2004]. On the other hand, work is still needed to make these systems "generally" applicable.

John McCarthy's 1971 Turing Award Lecture was titled "Generality in Artificial Intelligence." He later wrote [McCarthy, 1987]:

> It was obvious in 1971 and even in 1958 that AI programs suffered from a lack of generality. It is still obvious, and now there are many more details. The first gross symptom is that a small addition to the idea of a program often involves a complete rewrite beginning with the data structures. Some progress has been made in modularizing data structures, but small modifications of the search strategies are even less likely to be accomplished without rewriting.
>
> Another symptom is that no-one knows how to make a general database of common sense knowledge that could be used by any program that needed the knowledge. Along with other information, such a database would contain what a robot would need to know about the effects of moving objects around, what a person can be expected to know about his family, and the facts about buying and selling. This doesn't depend on whether the knowledge is to be expressed in a logical language or in some other formalism. When we take the logic approach to AI, lack of generality shows up in that the axioms we devise to express common sense knowledge are too restricted in their applicability for a general common sense database. In my opinion, getting a language for expressing general common sense knowledge for inclusion in a general database is the key problem of generality in AI.

The problems of modularity and generality, still very important today, have

not been well-addressed in the study of reasoning about actions.

In the field of programming languages, widely-used languages such as C++ [Stroustrup, 2000] and Java [Arnold *et al.*, 2000] support not only modularity in the form of simple inclusion of modules within each other, but also hierarchies of objects with inheritance of properties from parents. These languages have many libraries available for common use. Today's programmers would find it unimaginable to program without these features of modularity and generally available libraries.

On the other hand, work on reasoning about actions hasn't yet evolved out of that "unimaginable" state. Whenever we describe a new domain we start from scratch and state axioms that describe all of the effects of actions in detail. However, different actions in different domains are often related. For example, pushing, carrying, going are all special cases of moving an object from one place to another. Instead of describing all of their effects separately, it would be much easier and more convenient to describe the effects of moving once and then just add the domain-specific effects of each individual action.

The idea of having generally-applicable knowledge modules which can be used in the creation of more specialized knowledge is not new. Significant attempts have been made to build large databases of knowledge. The CYC project [Lenat and Guha, 1990, Matuszek *et al.*, 2006], under development for over 20 years, aims to create a very large database of commonsense facts. The KM Component Library [Barker *et al.*, 2001] is a collection of re-usable general purpose knowledge components. Both projects have their own representation languages and reasoning mechanisms, which are general in nature. Although these mechanisms can be used to reason about actions, the focus of the projects has been the development of the knowledge components, and the recent advances in reasoning about actions have not

been fully integrated. We would like to apply the idea of libraries of re-usable general purpose knowledge components [Barker *et al.*, 2001] to very expressive action languages.

The first prerequisite to being able to describe actions in terms of other actions is to develop methods of relating actions to each other. A preliminary step towards this was proposed in [Clark *et al.*, 1996], where components describing parts of STRIPS [Fikes and Nilsson, 1971] operators may be combined to form new operators. We propose a method of relating actions to each other, in the action language $\mathcal{C}+$ [Giunchiglia *et al.*, 2004]. This language has a rich set of features for describing action domains and has been applied to many domains [Lifschitz *et al.*, 2000, Lifschitz, 2000, Campbell and Lifschitz, 2003, Akman *et al.*, 2004].

Our work on describing actions as special cases [Erdoğan and Lifschitz, 2006] has led to the development of the Modular Action Description (MAD) language [Lifschitz and Ren, 2006]. This language allows writing several modules of action descriptions and then combining them, including the ability to build a hierarchy of action descriptions and inherit properties from parent modules. Its semantics is based on $\mathcal{C}+$. Having such a language, suitable for describing actions in terms of others, has enabled us to attempt McCarthy's idea in a restricted area: making a general library of commonsense knowledge about actions.

The library we constructed for this dissertation consists of MAD modules, each describing a group of general commonsense facts related to actions. For example, one module describes the effects of the "move" action, including the axiom "moving an object causes it to be at a new location." Another module expresses more general information about locality, such as "an agent must be at the same location as an object to be able to perform an action on it."

The original idea was to write the library in the MAD language, but as we worked on this, we saw much room for useful enhancements to MAD. Therefore, the library is actually written in an extended version of the MAD language. We also developed an implementation of our extended version of MAD in order to test our library modules and domain formalizations.

In deciding which modules to include in the library, we were guided by the large number of small domains that have been represented in $\mathcal{C}+$ by other researchers, as well as domains studied in other work on planning and reasoning about actions. Instead of trying to collect and formalize all actions one may encounter in a dictionary, we examined these previously-studied domains and factored out the common aspects, expressing them in a maximally general form. As the library was developed it became possible to obtain new formalizations of previously-studied domains by using library modules and only adding domain-specific facts.

Knowledge about actions is a small but important part of commonsense knowledge, and the research presented here is a small step towards solving the problem of generality in artificial intelligence. We plan to continue this work by building the library further, though such a library will remain a work-in-progress for a long time, due to the vast amount of commonsense knowledge in the world.

The layout of this dissertation is as follows. In the next chapter we provide some background on the state of the art in reasoning about actions, action languages and large databases of general-purpose knowledge. In Chapter 3 we review the action language $\mathcal{C}+$. Chapter 4 is the first chapter with our original work: it discusses our work on describing actions as special cases of other actions. After presenting that work which has greatly influenced the design of the MAD language, we take a step back in Chapter 5 to review the MAD language introduced by Lifschitz and

Ren [2006]. Chapters 6 and onwards present the rest of our original contributions. An overview of the enhancements we made to MAD during the course of our research is presented in Chapter 6.

Chapters 7–11 constitute the central part of this dissertation. The simple core of a database of action description modules is provided in Chapter 7 and then we show how some example action domains may be formalized using this core library in Chapter 8. The following chapters add more modules to the library: A theory of carriers is presented in Chapter 9, a library module about movement in regions is added in Chapter 10, and the usage of numbers is highlighted in library modules about time and resources in Chapter 11.

After the presentation of the library modules and examples, we go into some more technical issues: Chapter 12 describes further enhancements we made to MAD and the MAD implementation we developed is covered in Chapters 13 and 14. The former is about using the system; the latter covers some implementation details. Finally, in Chapter 15, we conclude and present some ideas for future work.

Chapter 2

Background

2.1 Reasoning about Actions

Automating the process of commonsense reasoning is among the goals of artificial intelligence. To accomplish this it is necessary to reason about actions and how they affect the states of the world. One of the earliest suggestions to dealing with commonsense reasoning was to use formal logic. Facts about the world can be represented as logical axioms and deduction methods can be used to reason about the changes in the states of the world [McCarthy, 1959].

Early research on logical reasoning about actions uncovered many problems. The "frame problem" [McCarthy, 1979] is the problem of describing what *doesn't* change as the result of an action. For example, when we move a block from one place to another, the locations of all of the other blocks remain unaffected. Typically an action changes only a few things which are easy to describe but it is infeasible to write axioms listing all the things that don't change. The "ramification problem" [Finger, 1986] is the problem of how to describe the indirect effects of actions. The main effect of an action is usually easy to describe because that's what we associate with

the action. However, there may be many consequences of that main effect which are not directly related to a specific action. For example, when a box is moved to a new location, all of the items in the box also end up at that new location. When there are such indirect effects of an action, the frame problem may become more difficult. Work on nonmonotonic reasoning and causality has produced formalisms which can solve both the frame problem [Shanahan, 1997] and the ramification problem [Geffner, 1990, Lin, 1995, McCain and Turner, 1997]. In particular, very expressive action description languages have been introduced, incorporating the solutions to these problems in their design. The development of action languages will be reviewed in the next section.

With the solution of important representational problems, the field of reasoning about actions has reached a certain level of maturity, where many examples can be studied and formalized successfully. Unfortunately, these examples are typically quite small and each one is formalized independently of the others. Concentrating on very small examples was a necessity for the development of precise formalisms that could solve the problems that arose. However, in order to address the issue of generality in AI that McCarthy [1987] brings up, we need to be able to go beyond small examples and extend our formalisms to make use of existing formalizations. There is ongoing research which aims to develop formalisms which address exactly this issue. We review some of this work in Section 2.3.

Expressive formalisms allowing re-use of existing action descriptions is only one half of solving the problem of generality for reasoning about actions. The other half that is necessary is a database of general-purpose action descriptions which can be used to quickly and conveniently describe new domains. As part of this dissertation we built such a database of action descriptions. Other researchers have

also worked on building databases of general-purpose knowledge, though none of these projects has focused specifically on reasoning about actions. We review this work in Section 2.4.

2.2 The Evolution of Action Languages

Action languages are formal models of parts of the natural language that are used for talking about the effects of actions [Gelfond and Lifschitz, 1998]. They define "transition systems" — directed graphs with the vertices representing the states of an action domain and the edges representing the transitions that are caused by performing actions (or by the passage of time). States are parameterized by "fluents"—propositions whose values change over time. For example, to represent whether a door is open or closed in a state, we may use a Boolean-valued fluent named *Open*.

One of the earliest systems invented to describe actions, STRIPS [Fikes and Nilsson, 1971], is closely related to the concept of an action language. The difference lies in the fact that STRIPS operators act on "world models," lists of first-order formulas, instead of states in a transition system. (World models may be incomplete in the sense that the values of some predicates may not be defined, whereas in a state of a transition system the values of all fluents are completely defined.) In STRIPS, preconditions of an action are stated in the form of first order formulas which should be entailed by the world model right before the action is to be executed, and the effects are stated as lists of formulas to be added to (those on the "add" list) or deleted from (those on the "delete" list) the world model right after the execution of the action. The frame problem is averted by the built-in assumption that the formulas not included in the delete lists remain true. The expressive capabilities of

STRIPS are limited. For example, it cannot represent action effects which depend on the situation (conditional effects), indirect effects of actions (ramifications), or the concurrent execution of actions. Also, many difficulties arise when trying to interpret the meaning of STRIPS descriptions because the semantics of the original language is not clearly defined [Lifschitz, 1987].

Pednault [1987] observed that the expressive power of STRIPS can be enhanced by allowing the add and delete lists to be conditional, and introduced the language ADL [Pednault, 1994]. This idea is shared by the action language $\mathcal{A}$ [Gelfond and Lifschitz, 1993], which has the same expressive power as the propositional fragment of ADL. (Although, strictly speaking, ADL also has fluents with non-Boolean values — "multi-valued" propositional fluents.)

Baral and Gelfond [1997] extended $\mathcal{A}$ by allowing concurrent execution of actions. Some reasoning systems try to deal with concurrency by "serializing" actions. While this works in some cases, there are situations where it really does matter whether actions are executed concurrently or in sequence. The example of the spacecraft Integer in [Lee and Lifschitz, 2003] illustrates this point: applying forces to the jets of a spacecraft along different axes will lead to the spacecraft being at different positions depending on whether the forces are applied concurrently or one after another.

Lin and Reiter [1994] and Baral [1995] observed that state constraints, traditionally used to represent the indirect effects of actions, actually correspond to multiple notions. While some constraints are about indirect effects others are about preconditions for actions, and it is necessary to distinguish between different kinds of state constraints. The need for causality in describing the indirect effects of actions was discussed in [McCain and Turner, 1995, Lin, 1995]. They argued that

the traditional way of representing state constraints as logical implications fails to capture the directionality of cause and effect. Turner [1997] introduced an action language which includes "static causal laws" for specifying causal relationships between fluents. Static causal laws do not mention actions but are useful for representing ramifications. As it turns out, such laws can also be used to specify implicit preconditions for actions (also called qualifications).

The idea of using causality was taken further when McCain and Turner [1997] introduced a logic of causal theories. These theories are very powerful, with the ability to express ramifications, qualifications, concurrency and nondeterminism. Instead of having a built-in assumption to solve the frame problem, it is possible to specify which fluents obey the "commonsense law of inertia." Such theories allow dynamic domains in which some fluents change by themselves (i.e. without actions occurring). Giunchiglia and Lifschitz [1998] turned causal logic into an action language, which they called $\mathcal{C}$.

Nonmonotonic causal theories were later extended to allow multi-valued fluents instead of only Boolean fluents, and the action language $\mathcal{C}+$ [Giunchiglia *et al.*, 2004] was introduced as an extension of $\mathcal{C}$. In addition to multi-valued fluents, $\mathcal{C}+$ introduced many other new features. Action attributes and defeasible causal laws may be used to make formalizations more "elaboration tolerant" [McCarthy, 2007], facilitating the modification of action descriptions by simply adding new statements, instead of changing existing ones. A special kind of fluent, called an "additive fluent" [Lee and Lifschitz, 2003] can be used to correctly calculate the aggregated effects of concurrent actions on numeric-valued fluents. (This kind of concurrency needs to be handled in a special manner.)

The Causal Calculator (CCALC)[1] is an implementation of a subset of $\mathcal{C}+$

[1] http://www.cs.utexas.edu/users/tag/cc/

(its "definite" fragment), which can be used to solve problems related to actions, such as prediction and plan generation. It has been applied to several challenging problems in commonsense reasoning [Lifschitz *et al.*, 2000, Lifschitz, 2000, Campbell and Lifschitz, 2003, Akman *et al.*, 2004], including domains of non-trivial size.

Recently, Lifschitz and Ren [2006] observed the need for an action language with the ability to refer to existing action descriptions. The Modular Action Description (MAD) language, based on $\mathcal{C}+$, allows action descriptions consisting of a list of modules, where a module mentioned earlier can be "imported" into a later module, possibly with some changes.

All of the languages above are action languages focusing on description of action domains. Even though implementations of them may have ways to represent certain queries, the languages don't have a formally defined way of representing queries. Baral and Gelfond [2000] introduce an action language $\mathcal{AL}$ that also has a history component $\mathcal{AL}_h$ for representing the happening and observation of actions, and a query component $\mathcal{L}_q$ for representing queries about the properties of the domain.

2.3 Modularity in Representing Actions

There have been several efforts at enhancing existing formalisms by adding modularity, similar to the emergence of MAD from $\mathcal{C}+$.

A paper by Gelfond [2006] is directed towards "the development and implementation of a library of knowledge modules needed for axiomatization of journey—a movement of a group of objects from one place (the origin) to another (the destination)." Adding modular structure to the logic programming language CR-Prolog [Balduccini and Gelfond, 2003, Balduccini, 2007] in that paper is similar to adding

modular structure to $\mathcal{C}+$ in [Lifschitz and Ren, 2006].

The applicability of the object-oriented paradigm to modeling dynamic domains is investigated by Gustafsson and Kvarnström [2004]. Their system is based on Temporal Action Logic [Doherty and Kvarnström, 2008]. The modularity comes from classes, associating a set of fluents and axioms with each object of that class. In contrast, modules in MAD are more general: they are essentially action descriptions, not focused on a particular class or object, though it is possible, in principle, to make MAD modules which mirror the notion of class.

The idea of adding modularity to logic programming has been explored in [Calimeri *et al.*, 2004]. They introduce "templates" as generic subprograms with some predicates used to parameterize the program, which can be viewed as similar to, but more restricted than, the constant renaming which happens in MAD. However, their examples are focused on "aggregates" in logic programming, rather than on commonsense reasoning or actions.

In [Baral *et al.*, 2006], the authors add modularity to answer set programming by using macros and "ensembles" (groups of macros). Then these macros can be used by replacing certain terms with others, or adding or removing terms during macro calls. They say that replacement is inspired by the original work on MAD in [Lifschitz and Ren, 2006]. The goals of this work are very similar to ours: to enable the creation of a library of knowledge modules. They provide examples of modules for planning and reasoning about actions. However, as far as we know, there have not been any attempts to use this methodology to build a library of knowledge modules, and currently there is no implementation of their macro call mechanism.

2.4 Large Databases of General-Purpose Knowledge

There have been several projects to build large-scale knowledge bases [Lenat and Guha, 1990, Knight and Luk, 1994, Fellbaum, 1998, Barker *et al.*, 2001], though none of these has focused specifically on reasoning about actions. In this section we will review two well-known efforts which also provide the ability to reason about actions: Cyc [Lenat and Guha, 1990] and the KM Component Library [Barker *et al.*, 2001]. After providing some general information about them we consider their facilities for reasoning about actions. Another comparison of the representation of actions in Cyc and KM may be found in [Parmar, 2001].

2.4.1 Cyc

Perhaps the best-known effort to create a very large repository of commonsense knowledge, the CYC project [Matuszek *et al.*, 2006, Lenat and Guha, 1990] was begun in 1984 and is still under development. As of March 2005, it contains approximately three million assertions interrelating over a quarter of a million concepts [Shepard *et al.*, 2005]. The knowledge base is written in the CycL language. This is a combination of a frame-based language [Minsky, 1975] and a constraint language. Most of the knowledge is expressed in frames but the constraint language, based on predicate calculus with higher order extensions, can be used to state things that are not possible to state with frames or even things about the frames themselves. It is reported that the main contribution of the higher-order extensions is to improve the efficiency of reasoning and that most of them (around 90%) can be transformed into first-order logic [Ramachandran *et al.*, 2005].

The ontology of Cyc contains units called "events," which have a starting time, ending time and duration [Lenat and Guha, 1990]. The class "action" is a

subclass of event, and actions are defined by what axioms and slots apply to them, just like any of the other frames in Cyc. For example, preconditions and effects of an action are represented in the slots of its frame. Events are the things that can be related to one another by temporal relations. They may be combined to make scripts. In order to use Cyc to reason about the effects of an event, we need to assert that the event happened. We can then use Cyc's inference engine, based on resolution, to prove or disprove statements related to that event. However, currently, this mechanism of Cyc does not address the frame problem even in the simple form that STRIPS uses. For example, we may assert that a plane flight from Austin to San Jose takes 3 hours and 44 minutes, that San Jose time is two hours behind Austin time, and that Frank flies from Austin to San Jose, leaving at 9:14 AM. Cyc will conclude that Frank is in San Jose at 10:58 AM. However, if we ask Cyc if Frank is in San Jose at 11:00 AM, it won't be able to prove or disprove it.

In addition to its main inference engine, Cyc also includes a planner [Shepard *et al.*, 2005], based on the SHOP planner [Nau *et al.*, 1999]. The representation of actions in this planner is very similar to STRIPS, with preconditions, add and delete lists. Therefore, reasoning about actions in a way that handles the frame problem in the built-in way that STRIPS does is possible for Cyc in the context of planning. However, even in just the planning context, it suffers from the representational shortcomings it inherits from STRIPS.

Another way to characterize Cyc is to say that it is "semi-formal." The syntax of the language used is formally defined, but the semantics is not. Therefore, the question of whether a reasoning process is "accurate" with respect to Cyc cannot be answered in a precise mathematical way.

2.4.2 The KM Component Library

Another project to build a large database of knowledge components is the KM Component Library [Clark and Porter, 1997, Barker *et al.*, 2001]. The emphasis of this project is very different from that of Cyc. The goal of the Cyc project is to collect a very large amount of commonsense facts, whereas the Component Library focuses on identifying repeated *patterns* of axioms in large theories and then abstracting them to form components. These components of axioms usually correspond to common English words. Another way in which the approach of the Component Library differs from Cyc is the purpose. Instead of being a knowledge base itself, the Component Library is intended to be a tool to build knowledge bases, enabling domain experts to easily and quickly build knowledge bases in their own fields. To this end the number of components is restricted to a few hundred and the number of relations between components to less than a hundred [Barker *et al.*, 2001].

The Component Library is written in the KM language [Clark and Porter, 2001b, Clark and Porter, 2001a], though the authors state that the approach is applicable to other languages. KM is frame-based but with many extensions. Frames group axioms in (mainly) first-order logic together. There are two main types of components in the library : entities (things that are) and events (things that happen). States and actions are both events. At first reading, it may be unexpected to see states be events. However, here a state is just a group of axioms representing a situation affected by actions.

KM has "situations" — representing the state of the world at a particular moment — to reason about changing states of the world and actions. The semantics of the situations is based on the Situation Calculus [McCarthy and Hayes, 1969],

where executing an action leads to a new situation. Unless we "enter" a specific situation, all the axioms and frames we write are part of the global situation, from which all the other situations inherit things. In KM frames slots values may be situation-dependent. Such slots are called "fluents" and there is a special kind of fluent, called an "inertial fluent," whose values persist from situation to situation. (i.e., the values remain the same after execution of an action, unless they are inconsistent with the rest of the situation.) By default all slots are inertial fluents.

Actions in KM are represented by a special class called "action" and the representation of actions is very similar to STRIPS operators (though the inference mechanism is quite different): actions have preconditions, negated preconditions, add lists, and delete lists. Unlike STRIPS, actions in KM may have situation-specific preconditions and effects. These are obtained by writing expressions to be evaluated in the current situation, instead of just formulas evaluating to true or false, as in STRIPS. Ramifications are also possible to represent in KM. Since things are computed upon query, if the value of a fluent is given as an expression to be evaluated based on the value of different fluents, after an action changes those fluents, a query about the first will be evaluated based on the new situation. It is recommended that fluents which are directly affected by actions be declared inertial fluents so that their values persist from situation to situation when the action doesn't affect them, and that fluents indirectly affected not be declared as inertial and that their values always be expressed in terms of other fluents. However, this assumes that fluents may be divided into two classes: those which are directly affected and those that are indirectly affected. Thielscher [1997] demonstrates that this may not always be the case and it is possible to have a fluent that may be directly affected sometimes and indirectly affected at other times. This limitation of KM is addressed

in the action languages $\mathcal{C}+$ and MAD, which we will be using in this dissertation.

Using KM situations, it is possible to simulate action executions or plans consisting of several action executions (though this may not always be straightforward since each action would need to provide a situation in which preconditions for the next action hold). We can then ask questions about what happens at the end or what holds in situations in-between. But other reasoning abilities such as planning or postdiction (asking questions about the past, given a history of events which may be incomplete) aren't currently available. Especially, projection backwards in time is not possible since the built-in way to solve the frame problem only projects the values of inertial fluents to situations forwards in time.

One interesting ability of KM is to reason about different paths of execution at the same time. This is a capability that is not available in the Causal Calculator. However, each of these paths is a path of single action execution. No concurrent execution of actions is allowed in KM.

Chapter 3

Prerequisites: $\mathcal{C}+$

In this chapter we provide part of the technical background necessary for our work. The library is written in an enhanced version of the MAD language [Lifschitz and Ren, 2006], the semantics of which is defined terms of the action language $\mathcal{C}+$ [Giunchiglia *et al.*, 2004]. Our review in this chapter introduces $\mathcal{C}+$. A review of MAD will be given in Chapter 5.

3.1 Multi-valued Signatures

A *(multi-valued) signature* is a set σ of symbols, called *(multi-valued) constants*, along with a non-empty finite set $Dom(c)$ of symbols, disjoint from σ, assigned to each constant c. The set $Dom(c)$ is the *domain* of c. A *Boolean* constant is one whose domain is the set $\{\mathbf{f}, \mathbf{t}\}$ of truth values.

Consider a fixed multi-valued signature σ. An *atom* is an expression of the form $c = v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *formula* is a propositional combination of atoms. If c is a Boolean constant, we will sometimes use c as a shorthand for the atom $c = \mathbf{t}$, and $\neg c$ as shorthand for $c = \mathbf{f}$. An *interpretation*

maps every constant in σ to an element of its domain. An interpretation I *satisfies an atom* $c = v$ if $I(c) = v$. The satisfaction relation is extended from atoms to arbitrary formulas according to the usual truth tables for the propositional connectives. A formula which is true under all interpretations is called *tautological*.

3.2 Action Descriptions

Consider a fixed multi-valued signature with the constants partitioned into three groups: *action* constants, *simple fluent* constants and *statically determined fluent* constants. A *fluent formula* is a formula such that all constants occurring in it are fluent constants, and an *action formula* is a formula that contains at least one action constant and no fluent constants.

An *action description* is a set of *(causal) laws*—expressions of the form

$$\textbf{caused } F \textbf{ if } G \tag{3.1}$$

or

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H. \tag{3.2}$$

where F, G and H are formulas satisfying certain syntactic conditions described below. Formula F is called the *head* of the law. A causal law is called *definite* if its head is an atom or $\bot$. If all the laws in an action description are definite, the action description is also called *definite*.

There are three types of causal laws. A *static law* is an expression of the form (3.1) in which F and G are fluent formulas. An *action dynamic law* is an expression of the form (3.1) in which F is an action formula and G is a formula. A *fluent dynamic law* is an expression of the form (3.2) in which F and G are

fluent formulas, H is a formula, and F does not contain any statically determined constants. We say that two causal laws are *similar* if they are of the same type.

Many useful constructs are defined as abbreviations for the basic forms (3.1) and (3.2) shown above. The reader is referred to [Giunchiglia *et al.*, 2004, Appendix B] for a detailed list. For instance, law

$$\mathit{DriveTo}(\mathit{Fred}, \mathit{Austin}) \textbf{ causes } \mathit{Location}(\mathit{Fred}) = \mathit{Austin}$$

stands for the fluent dynamic law

$$\textbf{caused } \mathit{Location}(\mathit{Fred}) = \mathit{Austin} \textbf{ if } \top \textbf{ after } \mathit{DriveTo}(\mathit{Fred}, \mathit{Austin}).$$

3.3 Transition Systems

According to the semantics of $\mathcal{C}+$, every action description D represents a transition system $TS(D)$—a directed graph whose vertices are states, and whose edges are labeled by events. States and events are interpretations of the fluent constants and action constants, respectively, that satisfy certain constraints determined by the causal laws in the action description. Thus to specify a state we assign a value to each of the fluents, and to specify an event we assign a value to each of the actions. Actions are usually Boolean, in which case we may view an event as a set of actions — those that are assigned the value true. If this set is empty, it means no actions are executed, and if it is a singleton, that means that there is no concurrent execution of actions in that event. Non-Boolean actions are used to represent action attributes, of which we will see an example in Section 4.2.

We said in the preceding section that the fluent constants are either sim-

ple or statically determined. Intuitively, statically determined constants represent properties that only depend on the values of other fluents in the same state, rather than being directly affected by actions executed during a transition.

In the next section we use an example to illustrate how a given action description corresponds to a transition system. The precise semantics is presented in Appendix A.

3.4 Example: the Suitcase Domain

The suitcase domain from [Lin, 1995] describes the spring-loaded locking mechanism of a suitcase with two latches, L_1 and L_2. The latches are either up or down and it is possible to toggle the position of each latch. The suitcase opens when both latches are open. The following $\mathcal{C}+$ action description formalizes this domain.

Notation: l ranges over $\{L_1, L_2\}$.

Simple fluent constants:	Domains:
$Up(l)$, $Open$	Boolean
Action constants:	Domains:
$Toggle(l)$	Boolean

Causal laws:

$$\textbf{inertial } Up(l), Open$$
$$\textbf{exogenous } Toggle(l)$$
$$Toggle(l) \textbf{ causes } Up(l) \textbf{ if } \neg Up(l)$$
$$Toggle(l) \textbf{ causes } \neg Up(l) \textbf{ if } Up(l)$$
$$\textbf{caused } Open \textbf{ if } \bigwedge_{l} Up(l)$$

The first causal law is an abbreviation for the fluent dynamic laws

$$\textbf{caused } \mathit{Up}(l) \textbf{ if } \mathit{Up}(l) \textbf{ after } \mathit{Up}(l)$$

$$\textbf{caused } \neg \mathit{Up}(l) \textbf{ if } \neg \mathit{Up}(l) \textbf{ after } \neg \mathit{Up}(l)$$

$$\textbf{caused } \mathit{Open} \textbf{ if } \mathit{Open} \textbf{ after } \mathit{Open}$$

$$\textbf{caused } \neg \mathit{Open} \textbf{ if } \neg \mathit{Open} \textbf{ after } \neg \mathit{Open}$$

It says that in the absence of any actions, the positions of the latches and the status of the suitcase (open or not) stay the same. The second causal law is an abbreviation for the pair of action dynamic laws

$$\textbf{caused } \mathit{Toggle}(l) \textbf{ if } \mathit{Toggle}(l)$$

$$\textbf{caused } \neg \mathit{Toggle}(l) \textbf{ if } \neg \mathit{Toggle}(l)$$

It expresses that a toggling action may happen or not happen at any time — its cause is exogenous. The third and fourth causal laws describe the direct effects of toggling actions and the last law states that the suitcase is open when both latches are up.

Figure 3.1 shows the transition system represented by this action description. There is an arc from each state to itself labeled $\emptyset$ (meaning no actions are executed) since all the fluents are inertial. Notice that there are no transitions from the middle state to any of the states on the left. This is because there are no actions that can make the suitcase closed once it is open.

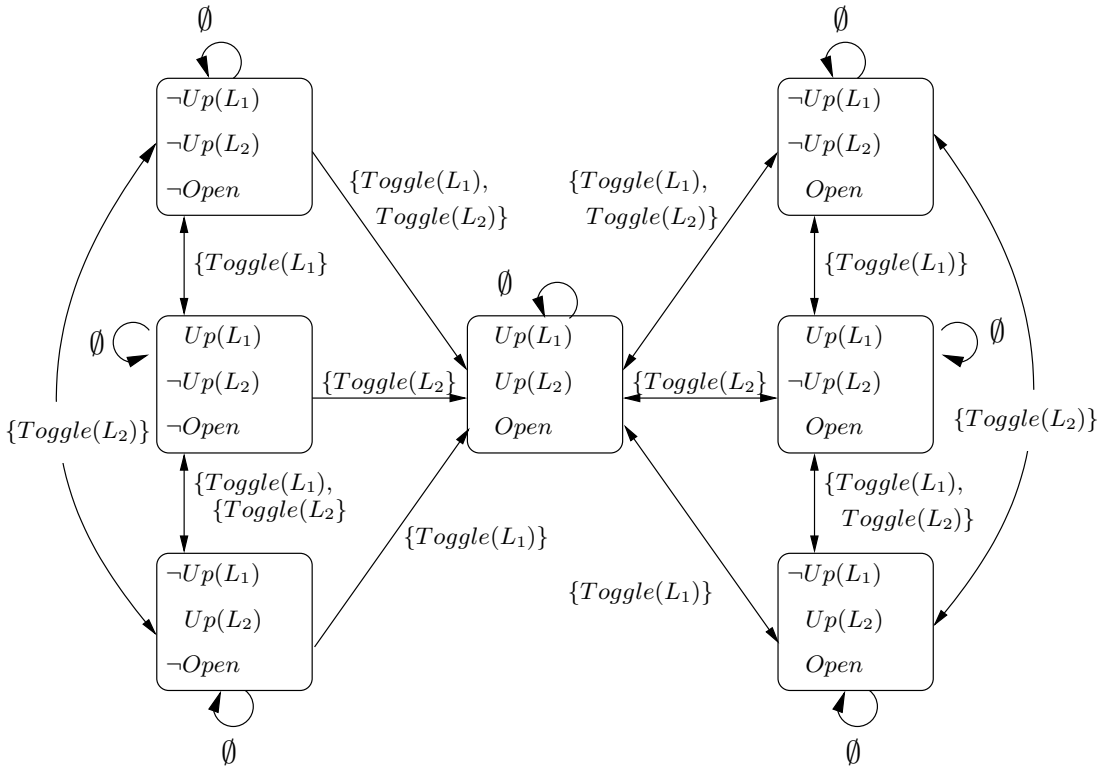

Figure 3.1: The transition system for the suitcase domain

3.5 The Causal Calculator (CCalc)

The Causal Calculator (CCalc)[1] is an implementation of the definite subset of $\mathcal{C}+$ which can be used to solve problems related to actions, including prediction, plan generation and postdiction. It is based on the idea of converting a problem to a set of clauses and using satisfiability solvers to find a solution. The idea of satisfiability planning comes from [Kautz and Selman, 1992]. Later McCain and Turner [1998] laid the basis for its use in CCalc. There has been some work in extending CCalc to cover nondefinite descriptions too. In particular, [Doğandağ *et al.*, 2004] defines a superset of definite action descriptions called "almost definite." The implementation in that work has the same input language as CCalc but uses a different computational mechanism.

Here is a formalization of the suitcase domain in the language of CCalc:

```
:- sorts
     latch.

:- objects
     l_1, l_2 :: latch.

:- constants
     up(latch), open :: simpleFluent(boolean);
     toggle(latch)   :: action(boolean).

:- variables
     L :: latch.

inertial up(L), open.
exogenous toggle(L).
```

[1] http://www.cs.utexas.edu/users/tag/cc/

```
toggle(L) causes up(L) if -up(L).
toggle(L) causes -up(L) if up(L).
caused open if [/\L | up(L)].
```

In contrast to $\mathcal{C}+$, CCalc has a specific syntax for defining multi-valued signatures (shown as the sort, object and constant declaration sections above). The causal laws are almost identical to the corresponding laws in $\mathcal{C}+$. One important difference is that the syntax of input files follows the Prolog tradition of capitalizing variables, since CCalc is written in Prolog. There are also a few changes due to being restricted to ASCII (such as using "`-`" for negation and "`[/\L | ... ]`" for finite conjunction over variable `L`).

CCalc allows declaring and using variables even though they are not part of $\mathcal{C}+$. Every proposition containing variables is treated as an abbreviation for a set of $\mathcal{C}+$ propositions. In a step called "grounding," CCalc replaces each variable (which is not in the context of a finite disjunction or conjunction) with every object of the corresponding sort.

The following is a query representing a planning problem for this domain. Initially both latches are down and the suitcase is closed. The query asks for the suitcase to be open in the final state.

```
:- query
   maxstep: 0..2;
   0: -up(l_1), -up(l_2), -open;
   maxstep: open.
```

Symbols `0:` and `maxstep:` are "time stamps". The time stamp `maxstep: 0..2` instructs `CCalc` to first try to find a plan of length 0, then 1, then 2, fail if no plan of these lengths exists.

CCalc responds as follows:

```
?- query 0.
% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 3 atoms, 1 clauses
% Writing input clauses... done.  (0.00 seconds)
% Calling GRASP... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0 seconds (prep 0 seconds, search 0 seconds)

No solution with maxstep 0.

% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 8 atoms, 13 clauses
% Writing input clauses... done.  (0.00 seconds)
% Calling GRASP... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0 seconds (prep 0 seconds, search 0 seconds)

0:

ACTIONS:  toggle(l_1)  toggle(l_2)

1:  up(l_1)  up(l_2)  open
```

This output shows that the shortest plan for this query takes one time step. None of the fluents are true initially, which is shown by the absence of anything after "0:". Actions `toggle(l_1)` and `toggle(l_2)` are executed concurrently between times 0 and 1, resulting in both latches being up and the suitcase being open at time 1.

Rigid constants in CCalc

One of the $\mathcal{C}+$ abbreviations defined in [Giunchiglia *et al.*, 2004, Appendix B] is for conveniently postulating that the value of a certain fluent always stays the same:

$$\textbf{rigid } c$$

stands for

$$\textbf{caused } \bot \textbf{ if } \neg(c = v) \textbf{ after } c = v$$

for all $v \in Dom(c)$.

CCalc augments the set of fluent constants by adding *rigid* constants. The declaration

```
:- constants
     c :: boolean;
```

denotes a Boolean-valued fluent constant `c` whose value will be the same at all times.

If the head of a causal law contains rigid constants, that causal law must not contain any non-rigid constants.

Chapter 4

Actions as Special Cases

The heart of every action language is a syntactic mechanism for describing effects of actions on fluents. When we define, for instance, the Monkey and Bananas domain in STRIPS, we can specify how pushing the box affects the location of the box by including appropriate atoms in the description of the operator $PushBox(l)$: we put $At(Box, l')$ for every location l' on its delete list, and $At(Box, l)$ on its add list. In $\mathcal{C}+$ the same idea can be expressed by the causal law

$$PushBox(l) \textbf{ causes } Loc(Box) = l \tag{4.1}$$

(quoted from [Giunchiglia *et al.*, 2004], Figure 2, reproduced in Section 4.3 below).

Descriptions like these are common in knowledge representation, but they are strikingly different from the descriptions of actions that humans give to each other informally. The dictionary says, for instance, that pushing is *moving by steady pressure*. This phrase explains the meaning of the word *push* not by listing the effects of this action, but by presenting it as a special case of another action, *move*, that is supposed to be already familiar to the reader. Some actions may need to be

described directly in terms of the changes that they cause; to move, for instance, means *to cause to change position*, according to the dictionary. But in most cases the easiest way to describe an action is to relate it to more basic actions.

Here is one more example of describing one action as a special case of another. A surgeon may indicate the action he wants to be performed by saying, "Scalpel." John McCarthy [1993] explains that in the context of an operation this one word may mean "Please give me the number 3 scalpel." The action to be performed is described as a special case of the basic action *give*.

In this chapter, we take some steps towards determining how a $\mathcal{C}+$ library of standard actions can be used when writing action descriptions. Specifically, we introduce a general form of causal laws for relating special-case actions and fluents to the library constants. These laws "explicitly define" a constant in terms of other constants. Therefore, we develop a theory of explicit definitions in $\mathcal{C}+$.

The laws used to define constants in terms of others can be called "bridge rules" because they provide a connection between the library and the specific domain description. With the theory of explicit definitions in $\mathcal{C}+$, bridge rules can be used, in certain cases, to completely eliminate all references to the library and thus obtain an alternative action description in terms of the domain-specific constants.

The causal laws used in the bridge rules are nondefinite. Since the Causal Calculator is an implementation of the definite fragment of $\mathcal{C}+$, it will not be possible to use that system, at least directly, to process action descriptions containing bridge rules. However, one of the propositions from our theory of explicit definitions shows that, under certain conditions, bridge rules may be replaced by definite causal laws.

In the rest of the chapter, a specific example is used to illustrate how action domains can be specified with libraries. First we give a $\mathcal{C}+$ description of the

action *Move* that can be included, in principle, in a library of general-purpose action descriptions. Then we review the formalization of the Monkey and Bananas domain from [Giunchiglia *et al.*, 2004] and show how to replace some of the $\mathcal{C}+$ laws in that formalization with a group of $\mathcal{C}+$ laws that characterizes *PushBox* as a special case of *Move*. It turns out that this reformulation is essentially equivalent to the original formalization. Finally, we demonstrate how this nondefinite reformulation involving the library may be turned into an equivalent definite action description.

4.1 Explicit Definitions in $\mathcal{C}+$

In classical logic, an explicit definition of a predicate constant P is an axiom of the form

$$P(x) \equiv \phi(x) \tag{4.2}$$

where ϕ is a formula that does not contain P. Such a definition has two properties. First, due to the equivalent replacement theorem of classical logic, if a theory contains axiom (4.2), any occurrences of P in other axioms may be eliminated. Second, adding axiom (4.2) to any theory which does not contain P yields a "conservative extension" of the original theory; any model of the new theory can be turned into a model of the original theory by dropping the predicate representing P.

Our goal is to develop a similar theory of explicit definitions in $\mathcal{C}+$.

An *explicit definition* of a multi-valued constant c, *in terms of* a multi-valued signature σ which does not contain c, is a set of causal laws of the form

$$\textbf{caused } c = v \equiv F_v \ , \tag{4.3}$$

one for each $v \in Dom(c)$, where

- each F_v is a formula of σ such that
  - if c is an action constant then F_v does not contain fluent constants;
  - if c is a statically determined fluent constant then F_v does not contain action constants;
  - if c is a simple fluent constant then F_v contains neither action constants nor statically determined fluent constants;
- the formulas

$$\bigvee_{v \in Dom(c)} F_v$$

  and

$$\bigwedge_{v,w \in Dom(c), v \neq w} \neg(F_v \wedge F_w)$$

  are tautological.

Intuitively, in view of the second condition, there is exactly one value of c corresponding to any interpretation of σ.

For example, the causal laws

$$\textbf{caused } \mathit{Clear} = L_1 \equiv (\mathit{Loc}(\mathit{Box}) = L_2 \vee \mathit{Loc}(\mathit{Box}) = L_3)$$
$$\textbf{caused } \mathit{Clear} = L_2 \equiv \mathit{Loc}(\mathit{Box}) = L_1$$
$$\textbf{caused } \mathit{Clear} = L_3 \equiv \bot$$

provide an explicit definition of the simple fluent constant *Clear* with domain $\{L_1, L_2, L_3\}$. Intuitively, *Clear* is the "first" location which is clear of the box.

The concept of an explicit definition in $\mathcal{C}+$, given above, differs from that in classical logic in two ways. In the case of classical logic a single formula ϕ suffices to define P since there are only two truth values. However, since $\mathcal{C}+$ is multi-valued, to define c we need a formula for each value in the domain of c. Another difference is the

restriction about the types of constants occurring in formulas F_v. Since language $\mathcal{C}+$ has three distinct types of constants, only certain constants may be used to define other constants. These differences notwithstanding, explicit definitions in $\mathcal{C}+$ are very similar to explicit definitions in classical logic, in that they share the replacement and conservative extension properties.

The following counterpart of the equivalent replacement theorem from classical logic allows us to eliminate all occurrences of an explicitly defined constant except its occurrences in the definition:

Proposition 1 *Let F, G be formulas, let D be an action description, and let L, L' be similar causal laws such that L' is obtained from L by replacing an occurrence of F by G. Then the action description*

$$D$$
$$L$$
$$\textbf{caused } F \equiv G$$

represents the same transition system as

$$D$$
$$L'$$
$$\textbf{caused } F \equiv G.$$

Proposition 2 below shows that adding an explicit definition of a new constant yields a "conservative extension." Let D and D' be action descriptions such that the signature of D is a part of the signature of D'. We say that D is a *residue* of D' if restricting the states and events of the transition system for D' to the signature of D establishes an isomorphism between the transition system for D' and the transition system for D.

Proposition 2 *Let D be an action description of a signature σ, and let c be a constant that does not belong to σ. If D' is an action description of the signature $\sigma \cup \{c\}$ obtained from D by adding an explicit definition of c in terms of σ, then D is a residue of D'.*

For instance, if D is an action description of a signature containing the fluent constant $Loc(Box)$, c is *Clear*, and D' is obtained from D by adding the explicit definition of *Clear* shown above, then the transition system for D' is isomorphic to that for D. The latter can be obtained by restricting the states of the transition system for D' to the fluent constants other than *Clear*.

Explicit definitions will play an essential role in relating special-case actions and fluents to actions and fluents in a general-purpose library. Such definitions constitute the "bridge rules" providing a connection between the library and the specific domain description. In such usage, an explicitly defined constant c may often appear in the heads of causal laws other than the definition, as part of one of the two action descriptions we wish to connect.

The causal laws used in explicit definitions are nondefinite because their heads are equivalences. In the general case, there is no known way to express definitions (with the two properties we would like them to have) using definite laws. However, if an action description does not refer to such a defined constant c in the heads of any laws other than the definition itself, then the definition may be equivalently expressed using definite causal laws:

Proposition 3 *Let σ be a signature and c be a constant that does not belong to σ. Let D be an action description of signature $\sigma \cup \{c\}$ which does not contain c in the heads of laws. Let D' be an action description of signature $\sigma \cup \{c\}$ obtained from D by adding an explicit definition (4.3) of c in terms of σ. Then the action description*

of signature $\sigma \cup \{c\}$ *obtained from* D *by adding the rules*

$$\textbf{caused } c = v \textbf{ if } F_v \qquad (v \in Dom(c))$$

represents the same transition system as D'*.*

The rest of the chapter focuses on an example of using a library description of action *Move* to reformalize the Monkey and Bananas domain from [Giunchiglia *et al.*, 2004].

4.2 Moving Things

Our "general-purpose" formalization of the action *Move* is a family of $\mathcal{C}+$ action descriptions depending on two parameters. For any nonempty finite sets P, L of symbols, the action description $MOVE(P, L)$ below represents the properties of moving physical objects (elements of P) to locations (elements of L).

This formalization uses action attributes, which are represented in $\mathcal{C}+$ as action constants which are non-Boolean. They have the special value *None* if the action of which they are an attribute is not executed.

The signature and the causal laws of $MOVE(P, L)$ are as follows:

Notation: p, p_1 range over P; l ranges over L.

Simple fluent constants:	Domains:
$Location(p)$	L

Action constants:	Domains:
$Move(p)$	Boolean
$Mover(p)$	$P \cup \{None\}$
$Destination(p)$	$L \cup \{None\}$

Causal laws:

$$\textbf{always } \mathit{Mover}(p) = \mathit{None} \equiv \neg \mathit{Move}(p) \tag{4.4}$$

$$\textbf{always } \mathit{Destination}(p) = \mathit{None} \equiv \neg \mathit{Move}(p) \tag{4.5}$$

$$\mathit{Move}(p) \textbf{ causes } \mathit{Location}(p) = l \textbf{ if } \mathit{Destination}(p) = l \tag{4.6}$$

$$\mathit{Move}(p) \textbf{ causes } \mathit{Location}(p_1) = l \textbf{ if } \mathit{Mover}(p) = p_1 \wedge \mathit{Destination}(p) = l \tag{4.7}$$

$$\textbf{nonexecutable } \mathit{Move}(p) \textbf{ if } \mathit{Location}(p) = \mathit{Destination}(p) \tag{4.8}$$

$$\textbf{nonexecutable } \mathit{Move}(p) \textbf{ if } \mathit{Mover}(p) = p_1 \wedge \mathit{Location}(p_1) \neq \mathit{Location}(p) \tag{4.9}$$

$$\textbf{exogenous } \mathit{Move}(p) \tag{4.10}$$

$$\textbf{exogenous } \mathit{Mover}(p)$$

$$\textbf{exogenous } \mathit{Destination}(p)$$

$$\textbf{inertial } \mathit{Location}(p) \tag{4.11}$$

The constants $\mathit{Mover}(p)$ and $\mathit{Destination}(p)$ are used here as attributes of the action $\mathit{Move}(p)$ in the sense of [Giunchiglia *et al.*, 2004, Section 5.6]. When the action $\mathit{Move}(p)$ is executed, the value of $\mathit{Mover}(p)$ is the agent executing that action, and the value of $\mathit{Destination}(p)$ is the location to which p is being moved; otherwise the value of each attribute is *None* ("undefined"). Executing $\mathit{Move}(p)$ causes the location of p and the location of $\mathit{Mover}(p)$ to be equal to $\mathit{Destination}(p)$. The action is not executable if $\mathit{Destination}(p)$ is the current location of p, and also if p and $\mathit{Mover}(p)$ are in different places.

Consider, for example, the transition system represented by the action description

$$\mathit{MOVE}(\{\mathit{Monkey}, \mathit{Box}, \mathit{Bananas}\}, \{L_1, L_2, L_3\}). \tag{4.12}$$

(This choice of "actual parameters," substituted for the "formal parameters" P, L, corresponds to the use of *MOVE* in the next section.) This graph has 27 vertices, corresponding to the states—assignments of locations L_1, L_2, L_3 to fluents

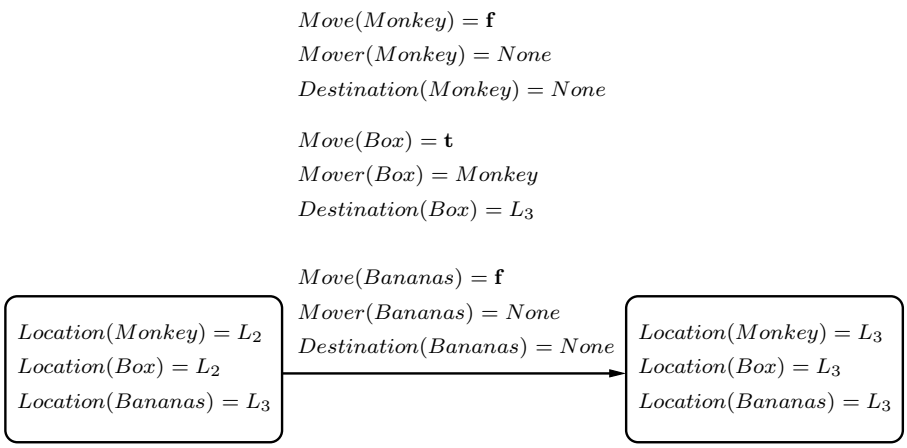

Figure 4.1: An edge of the graph represented by action description (4.12)

$Location(Monkey)$, $Location(Box)$ and $Location(Bananas)$. Every edge of this graph is labeled by an event—an assignment of values to the action constants. In one of these events, for instance, the monkey is moving the box from L_2 to L_3, where the bananas are. The corresponding edge of the graph is shown in Figure 4.1. Notice that even though the action $Move(Monkey)$ is not executed, the execution of action $Move(Box)$ with attribute $Mover(Box)$=$Monkey$ causes the location of the monkey to change.

4.3 Pushing the Box as a Special Case of Moving

The following action description, MB, was proposed in [Giunchiglia *et al.*, 2004, Figure 2] as a description of the familiar Monkey and Bananas domain. (Some of the causal laws are labeled for future reference.)

Notation: x ranges over $\{Monkey, Bananas, Box\}$; l ranges over $\{L_1, L_2, L_3\}$.

Simple fluent constants:	Domains:
$Loc(x)$	$\{L_1, L_2, L_3\}$

$HasBananas, OnBox$	Boolean

Action constants:	Domains:
$Walk(l), PushBox(l)$	Boolean
$ClimbOn, ClimbOff, GraspBananas$	Boolean

Causal laws:

$$\textbf{caused } Loc(Bananas)=l \textbf{ if } HasBananas \wedge Loc(Monkey)=l$$

$$\textbf{caused } Loc(Monkey)=l \textbf{ if } OnBox \wedge Loc(Box)=l$$

$$Walk(l) \textbf{ causes } Loc(Monkey)=l$$

$$\textbf{nonexecutable } Walk(l) \textbf{ if } Loc(Monkey)=l$$

$$\textbf{nonexecutable } Walk(l) \textbf{ if } OnBox$$

$$PushBox(l) \textbf{ causes } Loc(Box)=l \tag{4.13}$$

$$PushBox(l) \textbf{ causes } Loc(Monkey)=l \tag{4.14}$$

$$\textbf{nonexecutable } PushBox(l) \textbf{ if } Loc(Monkey)=l \tag{4.15}$$

$$\textbf{nonexecutable } PushBox(l) \textbf{ if } OnBox$$

$$\textbf{nonexecutable } PushBox(l) \textbf{ if } Loc(Monkey) \neq Loc(Box) \tag{4.16}$$

$$ClimbOn \textbf{ causes } OnBox$$

$$\textbf{nonexecutable } ClimbOn \textbf{ if } OnBox$$

$$\textbf{nonexecutable } ClimbOn \textbf{ if } Loc(Monkey) \neq Loc(Box)$$

$$ClimbOff \textbf{ causes } \neg OnBox$$

$$\textbf{nonexecutable } ClimbOff \textbf{ if } \neg OnBox$$

$$GraspBananas \textbf{ causes } HasBananas$$

$$\textbf{nonexecutable } GraspBananas \textbf{ if } HasBananas$$

$$\textbf{nonexecutable } GraspBananas \textbf{ if } \neg OnBox$$

$$\textbf{nonexecutable } GraspBananas \textbf{ if } Loc(Monkey) \neq Loc(Bananas)$$

$$\textbf{nonexecutable } Walk(l) \wedge PushBox(l)$$

$$\textbf{nonexecutable } Walk(l) \wedge ClimbOn$$

$$\textbf{nonexecutable } PushBox(l) \wedge ClimbOn$$

$$\textbf{nonexecutable } ClimbOff \wedge GraspBananas$$

exogenous $Walk(l)$
exogenous $PushBox(l)$ (4.17)
exogenous $ClimbOn$
exogenous $ClimbOff$
exogenous $GraspBananas$
inertial $Loc(x)$ (4.18)
inertial $HasBananas$
inertial $OnBox$

Action *PushBox* is a special case of *Move*, in which the object that is being moved is the box, the mover is the monkey, and the destination may be any one of the locations L_1, L_2, L_3. On the right margin we assigned numbers to the causal laws of *MB* that have counterparts in $MOVE(P, L)$. Our goal is to find a collection of causal laws ("bridge rules") relating *MB* to $MOVE(P, L)$ that will make (4.13)–(4.18) redundant. Causal laws (4.13) and (4.14), describing the effects of *PushBox*, will become "special cases" of (4.6) and (4.7), which describe the effects of *Move*. Causal laws (4.15) and (4.16), describing some of the preconditions of *PushBox*, will become redundant in the presence of the general preconditions (4.8) and (4.9) of *Move*. (The other precondition of the action *PushBox*—the fact that it cannot be executed if the monkey is on the box—is domain-specific and has no counterpart in the "library description" $MOVE(P, L)$.) Finally, (4.17) and (4.18) will become redundant in the presence of (4.10) and (4.11).

Our reformulation MB^* of *MB* is defined as follows. Its signature is the union of the signature of *MB* with the signature of the instance (4.12) of the "library description" of *Move*. Its causal laws are

- the causal laws of *MB*, except (4.13)–(4.18),

- the causal laws of (4.12), and
- the following causal laws, connecting (4.12) with MB:

$$\textbf{caused } \mathit{Location}(p) = \mathit{Loc}(p) \tag{4.19}$$

$$\textbf{caused } \mathit{Move}(\mathit{Box}) \equiv \bigvee_{l} \mathit{PushBox}(l) \tag{4.20}$$

$$\textbf{caused } \neg \mathit{Move}(p) \qquad (p \neq \mathit{Box}) \tag{4.21}$$

$$\textbf{caused } \mathit{Mover}(\mathit{Box}) = \mathit{Monkey} \equiv \mathit{Move}(\mathit{Box}) \tag{4.22}$$

$$\textbf{caused } \mathit{Destination}(\mathit{Box}) = l \equiv \mathit{PushBox}(l) \tag{4.23}$$

where p ranges over $\{\mathit{Monkey}, \mathit{Box}, \mathit{Bananas}\}$, and l over $\{L_1, L_2, L_3\}$.

Laws (4.19)–(4.23) are the bridge rules, connecting the domain-specific description (MB without laws (4.13)–(4.18)) with the library (4.12). Causal law (4.19) says that *Location* is synonymous with *Loc*. Laws (4.20) and (4.21) tell us that moving the box amounts to pushing it to some location, and that no object other than the box is ever moved. According to (4.22), the mover is the monkey whenever the box is being moved. According to (4.23), the destination is l whenever the box is pushed to l.

We mentioned earlier that our bridge rules would take the form of explicit definitions. Specifically, every bridge rule defines a constant from a library instance in terms of the domain-specific signature. For example, the first, (4.19) is in fact equivalent to

$$\textbf{caused } \mathit{Location}(p) = l \equiv \mathit{Loc}(p) = l \qquad (l \in \{L_1, L_2, L_3\}) \tag{4.24}$$

which is an explicit definition of *Location* in terms of the signature of MB. Upon inspection of (4.20)–(4.23) we see that they look quite similar to explicit definitions of the constants from (4.12) in terms of the signature of MB, although they don't

exactly match the pattern of definitions as characterized in Section 4.1. It was more convenient to express these rules as shown. However, using Proposition 1 and some other propositions which allow for modification of action descriptions while preserving the corresponding transition systems, it is possible to turn (4.19)–(4.23) into explicit definitions.

It is interesting to note that the bridge rules are explicit definitions of library constants in terms of the signature of MB, and not the other way around. This is somewhat surprising, since we planned to use the library of basic actions to describe the domain-specific actions as special cases of actions in the library. On the other hand, the explicit definitions in the bridge rules can be considered as stating exactly which special case of the library action corresponds to the domain-specific events.

Another way to view this is in analogy with the use of concepts of abstract algebra in the definition of a specific number system. When we describe the set $\mathbf{R}$ of real numbers as a group relative to addition, with the neutral element 0, we say essentially that the axioms for groups

$$\begin{array}{ll} \forall x, y, z \in G & x \star (y \star z) = (x \star y) \star z, \\ \forall x \in G & x \star e = x, \\ \forall x \in G \, \exists y \in G & x \star y = e \end{array}$$

hold if

$$\begin{array}{l} G \text{ is } \mathbf{R}, \\ \star \text{ is } +, \\ e \text{ is } 0. \end{array}$$

Just like our bridge rules, here we specify how the "library concepts" should be replaced by special cases.

Action description MB^* is not exactly equivalent to MB, because its signa-

ture is different. A state of MB assigns values to the fluent constants

$$Loc(p), HasBananas, OnBox;$$

a state of MB^* assigns values to all these constants and also to $Location(p)$. An event of MB assigns values to the action constants

$$Walk(l), PushBox(l), ClimbOn, ClimbOff, GraspBananas;$$

an event of MB^* assigns values to the all these constants and also to

$$Move(p), Mover(p), Destination(p).$$

The proposition below shows, however, that the transition systems represented by MB and MB^* are isomorphic to each other. In this sense, our reformulation of MB based on the "toy library" is adequate.

Proposition 4 *MB is a residue of MB^*.*

The proof of this proposition, outlined in the appendix, relies on Propositions 1 and 2. However, note that to use Proposition 2, (i) the extended action description must contain only explicit definitions, and (ii) the action description which is a residue must not contain the constants being explicitly defined. We had stated earlier that it is possible to turn (4.19)–(4.23) into explicit definitions. This gives us an action description satisfying condition (i). To satisfy the second, we may modify MB^* without altering its transition system, by first using Proposition 1 and the definitions to replace all constants in (4.12) by formulas of the signature of MB.

Whenever we have an action description containing bridge rules that explic-

itly define all constants from the library (such as MB^*), we may obtain a residue for it that does not contain the library constants, in two steps. First, we apply Proposition 1 to turn the causal laws coming from the library into equivalent laws involving only domain-specific constants. Then the bridge rules will be the only laws referring to the library. Second, we drop the bridge rules to obtain an action description for the domain, which doesn't refer to the library at all. By Proposition 2, this new description will be a residue. Applying this procedure to MB^* will yield an action description which has the same transition system as MB.

4.4 Turning MB^* into a Definite Theory

We have shown that the formalization of the Monkey and Bananas domain may be reformulated using our "toy library" $MOVE(P, L)$, as the nondefinite action description MB^*. As discussed in the introduction to this chapter, from an implementation point of view it is important to be able to turn a nondefinite action description into a definite one. Here we show how to do this for MB^*.

The first nondefinite causal law in MB^* is (4.19), which is equivalent to (4.24). We would like to use Proposition 3 to make it definite. However, Proposition 3 is not directly applicable because laws (4.6), (4.7) and (4.11) contain *Location* in their heads. Therefore we first use Proposition 1 in the presence of (4.19) to replace *Location* by *Loc* in the heads of (4.6), (4.7) and (4.11). Now we may use Proposition 3 to replace (4.19) with the definite causal laws

$$\textbf{caused } \mathit{Location}(p){=}l \textbf{ if } \mathit{Loc}(p){=}l \quad (l \in \{L_1, L_2, L_3\}).$$

The remaining nondefinite laws (4.20)–(4.23) contain only action constants.

They may be transformed into definite laws using Proposition 5 below.

Recall that one of the abbreviated causal laws in the suitcase example on page 23 stated that the action *Toggle(l)* was exogenous — its value could be true or false at any time. A constant c is said to be *exogenous* in an action description D if the action description contains the causal laws

$$\textbf{caused } c{=}v \textbf{ if } c{=}v$$

for all values $v \in Dom(c)$.

Proposition 5 *Let D be an action description and F be a formula such that all constants in F are action constants which are exogenous in D. Then*

$$D$$
$$\textbf{caused } F \textbf{ if } G$$

represents the same transition system as

$$D$$
$$\textbf{caused } \bot \textbf{ after } \neg F \wedge G.$$

This proposition is often applicable to a causal law containing only action constants, such as (4.20)–(4.23), because action constants are usually exogenous. For instance, in the presence of (4.10), we can replace (4.21) with **caused** $\bot$ **if** $Move(p)$.

Chapter 5

Prerequisites: MAD

Our library of action descriptions is written in an extended version of MAD, a modular action language based on $\mathcal{C}+$. MAD is heavily influenced by our work on representing actions as special cases, presented in the previous chapter. In this chapter we review the original syntax and semantics of MAD, which was introduced in [Lifschitz and Ren, 2006], before presenting our extensions to MAD in the next chapter.

We begin this chapter with an overview of the syntax of MAD, illustrated by an example, followed by an overview of the semantics. After that we provide details on the syntax and semantics of MAD's distinguishing feature, import statements.

5.1 Syntax Overview

A MAD action description consists of several modules $M_1; ...; M_n$ (separated by semicolons) with the possibility of later modules referring to earlier ones.

The syntax of MAD draws upon that of CCalc and $\mathcal{C}+$. A MAD module consists of a name and the following parts: a sort declaration part, an object

declaration part, a constant declaration part, a variable declaration part, and axioms. None of the declaration parts or axioms are required in a module, though, if they appear, it must be in the order given above. In addition, in between any of these declaration and axiom parts there may be any number of *import* statements, referring to other modules (which occur earlier in the sequence of modules). Here we only consider the simplest kind of imports. The detailed structure of import statements will be shown in Section 5.4.

Like in CCalc, the declarations serve to specify the signature in which we write the axioms. The axioms are expressions similar to causal laws in the sense of $\mathcal{C}+$. The list of axioms can be thought of as a $\mathcal{C}+$ action description (and actually becomes a $\mathcal{C}+$ action description after grounding the variables).

5.2 Example: the Suitcase Domain

Here is a MAD module which is very similar to the $\mathcal{C}+$ description of the suitcase domain given in Section 3.4.

module SUITCASE;
  sorts
    Latch;
  constants
    Up(*Latch*), *Open*: **fluent**;
    Toggle(*Latch*) : **action**;
  variables
    l: *Latch*;
  axioms
    inertial *Up*(*l*), *Open*;
    exogenous *Toggle*(*l*);
    Toggle(*l*) **causes** *Up*(*l*) **if** ¬*Up*(*l*);
    Toggle(*l*) **causes** ¬*Up*(*l*) **if** *Up*(*l*);

Open **if** $\forall l\ Up(l)$;
endmodule

All constants (two fluents and one action) in this example are Boolean-valued. If we wanted to declare a fluent constant with the domain being a sort other than Boolean, we would specify that by putting that sort in parentheses after the word **fluent** in the constant declaration section.

Notice that the module above does not contain an object declaration part. In this form it does not correspond to any $\mathcal{C}+$ action description. In order for a MAD action description to have a model, each of the sort names occurring in it must be characterized by finite, nonempty sets of symbols.

We may add a second module

module TWO_LATCHES;
  import SUITCASE;
  objects
    L_1, L_2: *Latch*;
endmodule

which imports module SUITCASE and assigns the set of objects $\{L_1, L_2\}$ to the sort *Latch*. By concatenating these two modules we may create the action description

SUITCASE; TWO_LATCHES

which describes suitcases that have two latches, L_1, L_2. This two-module action description is equivalent to the $\mathcal{C}+$ action description we saw in the previous chapter.

5.3 Semantics Overview

The semantics of MAD is defined by translating MAD into $\mathcal{C}+$. It is composed of two parts: A MAD action description (possibly containing many modules) is first turned into a single-module action description which is considered to have the same meaning. Then this single-module action description is turned into an action description in $\mathcal{C}+$.

As we did in the review of $\mathcal{C}+$, we use an example to illustrate the process of turning a MAD action description into a $\mathcal{C}+$ description. The precise semantics is presented in Appendix B.

Consider turning the two module action description

SUITCASE; TWO_LATCHES

into an equivalent single-module action description. Here is the result:

module TWO_LATCHES;
  sorts
    Latch;
  objects
    L_1, L_2: *Latch*;
  constants
    Up(*Latch*), *Open*: **fluent**;
    Toggle(*Latch*) : **action**;
  variables
    I1.l: *Latch*;
  axioms
    inertial *Up*(I1.l), *Open*;
    exogenous *Toggle*(I1.l);
    Toggle(I1.l) **causes** *Up*(I1.l) **if** $\neg$ *Up*(I1.l);
    Toggle(I1.l) **causes** $\neg$ *Up*(I1.l) **if** *Up*(I1.l);

Open **if** $\forall$I1.l Up(I1.l);
endmodule

The contents of imported module SUITCASE are incorporated into the importing module TWO_LATCHES. During this process, all variables in module SUITCASE are renamed. Since l is the only variable, all occurrences of it are modified by appending "I1." to them. The prefix "I1." stands for "import number 1" because it is due to the first (and only) import statement in the action description.

5.4 MAD import Statements

In the previous chapter we showed how it is possible to use explicit definitions to define fluent and action constants in terms of other fluents and actions, and we provided an example of how this method can be used to refer to a library action description from within a domain-specific description. Our work described in that chapter has influenced the syntax and semantics of MAD import statements proposed by Lifschitz and Ren [2006].

The import statement we saw in the TWO_LATCHES module in Section 5.2 was very simple. It only contained the name of the module to be imported. In general, an import statement can contain, in addition to the name of the module to be imported, sort renaming clauses and constant renaming clauses. The general form is

import *NAME*;
s_1 **is** s'_1;
...
s_k **is** s'_k;
$c_1 \cdots$ **is** F_1;
...

$c_l \cdots$ **is** F_l;

where *NAME* is a module name, $s_1, ..., s_k, s'_1, ..., s'_k$ are sort names, $c_1, ...c_l$ are constant names, and $F_1, ..., F_l$ are formulas . The dots after each c_j represent the possibility of having variables as the arguments and domain of constants. There are some more restrictions on the exact form of the constants and formulas appearing in **is** clauses, which are explained in Appendix B. These conditions on the syntax of constant renaming **is** clauses are similar to the conditions given in import statements Section 4.1 for an explicit definition.

For example, here is another version of module TWO_LATCHES that has a more complicated import statement, which may be used to assign different names to sorts and actions in the imported module:

module TWO_LATCHES;
  sorts
    Lock;
  objects
    L_1, L_2: *Lock*;
  constants
    Right, *Left*: **action**;
  variables
    l : *Lock*;
  import SUITCASE;
    Latch **is** *Lock*;
    Toggle(l) **is** $(Right \wedge l = L_1) \vee (Left \wedge l = L_2)$;
endmodule

In the module above we have a new sort *Lock* (a special kind of latch) with the two latches being of this sort. We also have two new actions, *Right* and *Left*,

which we would like to have as short names for toggling locks L_1 and L_2, respectively, similar to the surgeon's "Scalpel" in the introduction of this chapter.

Clauses following the name of the module to be imported are instructions on how the imported module should be modified before incorporating it into the new module. The first **is** statement above is a sort renaming clause. It says that every occurrence of sort *Latch* in the module SUITCASE should be replaced by *Lock*. The second **is** statement is a constant renaming clause. It provides an explicit definition of *Toggle* in terms of actions *Right* and *Left*. It says that the equivalence

$$\mathrm{I1}.Toggle(\mathrm{I1}.l) \equiv (Right \wedge \mathrm{I1}.l = L_1) \vee (Left \wedge \mathrm{I1}.l = L_2) \tag{5.1}$$

should be added to the module and that all occurrences of *Toggle* should be replaced by I1.*Toggle* in order to indicate that *Toggle* has been defined in terms of other constants now. (Constants which have not been redefined keep their original names.)

Axioms introduced by importing, such as causal law (5.1), are similar to the equivalences we saw in bridge rules (4.19)–(4.23) earlier. Like those, the equivalence has a constant, coming from the "library," on the left-hand side, and a formula, in terms of constants from the specific domain, on the right-hand side.

When discussing the equivalences we saw in bridge rules (4.19)–(4.23) earlier, in Section 4.3, we drew an analogy with the use of concepts of abstract algebra in the definition of a specific number system. By specifying that the symbols G, $\star$, e should be replaced by $\mathbf{R}$, $+$, 0, respectively, we "inherited" the axioms in a modified form. Similarly, the import statement from the module TWO_LATCHES,

import SUITCASE;
Latch **is** *Lock*;
Toggle(l) **is** $(Right \wedge l = L_1) \vee (Left \wedge l = L_2)$;

tells us that everything called a latch in library module SUITCASE is now a lock, action *Right* has all properties that are postulated for the action $Toggle(L_1)$ in the library module SUITCASE, and similarly for the action *Left*. For example, with this import the axiom

$$\textit{Toggle(l)} \textbf{ causes } \textit{Up(l)} \textbf{ if } \neg \textit{Up(l)};$$

from module SUITCASE (where l is a variable for latches) has the same effect as if we had written the axioms

$$\textit{Right} \textbf{ causes } Up(L_1) \textbf{ if } \neg Up(L_1);$$
$$\textit{Left} \textbf{ causes } Up(L_2) \textbf{ if } \neg Up(L_2);$$

in module TWO_LATCHES.

Multiple Imports

In the study of explicit definitions and their use in bridging, we were concerned with using only one copy of the library about moving. In practice it is essential to be able to use the same general-purpose action description from the library multiple times in a new action description. For instance, of the actions in *MB*, three others besides *PushBox* can be expressed as special cases of *Move*. The actions *Walk*, *ClimbOn* and *ClimbOff* may be viewed as the monkey moving itself.

MAD includes the capability of importing several copies of the same module into another module. This is the source of the "I1." appended to the front of variables and to constants which have been explicitly defined. The 1 in "I1." indicates that the renamed variables and constants come from the first instance imported.

In the case of more than one copy being imported, variables and constants are renamed by appending "Im." where the integer m is the smallest integer which does not occur in the MAD action description.

Chapter 6

Enhancing the MAD Language

In the preceding chapter we showed how the MAD language enables us to use import statements to define fluent and action constants in terms of other fluents and actions. Our initial plan was to use the MAD language with its import facility for building a library of general-purpose action descriptions. However as we worked on this, we identified many new features that we would like to have in our language and also many ways in which some parts of the language definition should be tweaked. Here we give an overview of the enhancements that we made to the syntax of the MAD language. Further enhancements are presented in Chapter 12.

A Note on Fonts

When discussing $\mathcal{C}+$ and MAD in the previous chapters, we used various **bold** and *italic* fonts to write parts of action descriptions. Starting in this chapter we switch to `typewriter` font for everything written in our enhanced MAD language, because now we are showing a language which has an actual implementation. (We took a similar approach when discussing CCalc, however the implementation of

the enhanced version of MAD is not implemented in Prolog, and thus does not have some of the restrictions of CCalc, such as having to use "`:-`" before declarations and everything capitalized having to be a variable.)

New Features

Several new features were added to MAD. The full description of what is allowed in the enhanced syntax is included as Appendix D.

- *Built-in sort* `action`: In order to express properties of actions, we extended MAD with a new, built-in sort, `action`. All constants declared as actions become objects of this sort. This sort can then be used as the argument of a fluent constant. For example, a Boolean fluent constant declared as

  ```
  constants
     Executed(action) : fluent;
  ```

  may denote whether an action is executed or not.

  The original syntax of MAD allows only identifiers as arguments to action or fluent constants appearing in axioms. This feature also necessitates having constant arguments which have arguments themselves — action names with arguments.

  In order to postulate axioms about actions in general, not just about specific actions, we introduced action variables. These may occur in axioms whenever an action constant may occur.

- *Built-in sort* `explicitAction`: MAD and $\mathcal{C}+$ allow concurrent execution of actions but sometimes we want to prevent concurrency. A convenient way to express this is to write an axiom such as

```
nonexecutable a & a1 if a!=a1
```

where `a` and `a1` are variables ranging over the declared action names. Axioms of this form have been used successfully in CCALC, where there are no modules or `is` statements. However, in MAD, we can use `is` statements with constant renaming clauses, to rename actions. The problem with this is that whenever an action is renamed in terms of another, the original action name is preserved (albeit prepended with a prefix of the form "Im.", where m is an integer) along with the new name, and the semantics of MAD guarantees that both the names for the action are assigned the same values (true when executed, false otherwise). Therefore including this axiom eliminates all models in action descriptions with at least one action renaming. What we would really like to accomplish is to prevent the concurrent execution of actions that have not been renamed. In order to distinguish such "explicitly declared" actions from those which are "implicitly declared" through renaming statements, we introduced a new built-in sort `explicitAction`, the objects of which are all action names which do not have a prefix of the form "Im." (after conversion to a single module). Using variables for `explicitAction` in the axiom above solves the problem.

- *Sort inclusion declarations*: Sometimes it is convenient to define relations between sorts, such as the subset relation. Sort inclusion declarations serve to express when every object of a certain sort s_1 is also a sort of s_2. For example,

```
inclusions
  Agent << Thing;
```

states that every object of sort *Agent* is also of sort *Thing*.

- *Moving sort declarations out of modules*: A common problem we ran into when trying to write modular action descriptions was the need to repeat sort declarations before importing modules. Imagine we are writing a new module M (which will import module M') and want to refer to some sorts which have already been declared in M'. If we don't need to rename any constants during the import, it is possible to simply import M' at the beginning of module M, and refer to the sorts in M' afterwards. However, if we need to rename some constants during the import, we need to declare constants and variables in M before importing M', and for that we need to declare the sorts that will be used in the constant/variable declarations. This situation forced us to declare the same sorts in many modules.

  To fix this issue, we now move sort and sort inclusion declarations outside of individual modules, and make them sections on the order of modules. An action description is a sequence of not just modules but sort declarations, inclusions and modules. Sorts and sort inclusions are treated as if they are declared in all the modules following them.

  Now, instead of having library modules each with its own sorts and inclusions, we can have a library "ontology" alongside the library.

- *Objects with arguments*: We allow objects with arguments. This allows us to define objects which are associated with other objects. For example,

```
sorts
  Building; Person;

objects
  John, Bob     : Person;
  House(Person) : Building;
```

One thing that is important when we have objects with arguments is to make sure that this doesn't lead to infinite grounding. For example,

```
sorts
  Person;

objects
  John, Bob      : Person;
  Friend(Person) : Person;
```

would cause there to be an infinite number of `Person` objects. In order to guarantee such things don't happen, the implementation introduces sort dependencies and certain restrictions on how to declare objects with arguments. The details of this will be given Chapter 12.

- *Integer ranges as built-in sorts and integers as built-in objects*: Oftentimes it is convenient to talk about an "array" of objects which are indexed by an integer. For example, if we want to formalize a soccer-playing domain, we might like to write

  ```
  objects
    Player(1..11) : Person;
  ```

  to declare the players of one team. Or we might want to have a fluents with integer domains. For example,

  ```
  sorts
    Event;

  fluents
  ```

```
Hour(Event) : 1..24;
Minute(Event) : 1..60;
```

In order to allow such declarations, we added integer ranges as built-in sorts. They may appear as arguments for objects/constants and as the domain for fluents. Variables can have an integer range as their sort too. However, integer ranges may not be declared as sorts or inclusions, and they can't be part of a sort renaming clause in an import statements.

Integers are added as built-in objects and may appear anywhere a declared object can.

- *Basic arithmetic and comparison*: Once we have integers, then we can go beyond simple atomic formulas of the form `c=v` or `c!=v`. We can also compare values of integers. For example,

  ```
  nonexecutable spend(m) if MyMoney < m;
  ```

  We enhanced MAD by adding the comparison operator "`<`" and arithmetic operators "`+`" and "`*`".

  The comparison operator "`<`" cannot occur in the head of axioms.

  The "`+`" and "`*`" operators may not appear as arguments to constants or objects. Instead, they may be used by introducing an additional variable. E.g.,

  ```
  value(z)=k if index=x & offset=y & z=x+y;
  ```

  must be used instead of

  ```
  value(x+y)=k if index=x & offset=y;
  ```

- *Numeric symbols*: As in all high-level programming languages, having symbolic constants is useful. In order to make integer-related descriptions general, we introduced numeric symbols. For example, we can write

```
numeric_symbol  num_cars=15

...

module TRAFFIC;

  objects
    Car(1..num_cars) : Vehicle;
```

  This tells the system to treat all occurrences of the string "`num_cars`" as `15`.

  With this capability, we can write the library modules using numeric symbols and the user of the library can create an extra file defining his preferred integer values for those symbols, to be given to the system in advance of the library ontology and the library.

- *Extra variables in constant renamings*: Sometimes it is convenient to allow the right-hand side of a constant renaming `is` statement to contain new variables that do not occur in the left-hand side. For example, if we already have an action of an object `x` being moved to a place `p`, and want to define a new action of an agent `u` carrying `x` to `p`, we can use the following `import` section:

```
import MOVE;
  Move(x, p) is Carry(u, x, p);
```

  The intuitive meaning of this is "the action of agent `u` carrying object `x` to place `p` is synonymous with the action of moving `x` to `p`." Some properties of

`Carry` with respect to `x` and `p` are inherited from `Move` but the properties with respect to `u` need to be defined additionally.

In the importing of MOVE above, the right-hand side of the `is` statement has more variables than the left-hand-side. The variables that do not occur in the left-hand-side can be thought of as implicitly quantified over the import statement. In such cases, what is imported is not one copy of *Move* with the name I1.*Move*, but several copies, each with a distinct name. This requires a change to the semantics, which will be detailed in Chapter 12.

- *Separate action and fluent constant declaration sections*: Instead of having one section declaring both action and fluent constants, now we have two sections: one for actions and one for fluents.

- *Sort name predicate shorthand*: Sometimes it is convenient to write a formula stipulating that a variable or object belongs to a certain sort. For example, we may write

```
constraint Support(x)=y -> exists c c=y;
```

where `y` and `c` are variables for sorts `Supporter` and `Carrier`, respectively, and the latter is a subsort of the former. The intuitive meaning of this is "the object `y` which supports thing `x` must also be a carrier."

In order to make such formulas more convenient to express, we allow a shorthand where sort names appear as unary predicates. Now the axiom above may be written as:

```
constraint Support(x)=y -> Carrier(y);
```

Such shorthand predicates are easily translatable to CCalc.

- *Include statements*: The original definition of MAD requires an action description to be a sequence of modules separated by semicolons. In our implementation we allow `include` statements (similar to inclusion statements in many programming languages) in place of modules at the beginning of action descriptions. Such include statements refer to another file containing an action description, to be inserted in place of the `include` statements. Inclusions may also be nested, meaning an included file may have include statements itself.

Chapter 7

The Core Library

In Chapter 4, we conjectured that a library of standard descriptions for a number of "basic" actions can facilitate writing, understanding and modifying action descriptions, and illustrated this idea by presenting a toy movement library, and showing how the action *PushBox* in the Monkey and Bananas domain can be described as a special case of the "library action" *Move*.

In this chapter we present a simple core for a library with which we can formalize some classic domains from the literature on commonsense reasoning. Later chapters will add more elaborate modules to the library to expand the scope of domains we can formalize.

Most of the library modules presented in this chapter are "general" action descriptions — not about any specific actions. There are also a couple of modules about specific fluents and actions: one about objects being moved from place to place, and the other about objects being supported by others.

Recall from Chapter 6 that sort and inclusion declarations now appear outside of modules. Any modules following these declarations are treated as if the

declarations appear in the modules. Because of this we structure our library in two parts:

- a library "ontology", specifying all the sorts and inclusion relationships for the library modules
- the library modules, without any sort or inclusion declarations

In this chapter we'll show each sort and inclusion declaration from the ontology right before the first library module that needs it. Then at the end of the chapter we'll present all of the ontology together.

The ontology file must always be included before the library modules. (Usually by an `include` command at the beginning of a file.)

Example

Here is an example of a MAD file formalizing the blocks world, which we will study again in Section 8.1. The first two lines below are instructions to include the file containing the library ontology and the file with the library modules, which we will cover in detail in this chapter. The third line says that all occurrences of symbol "`MaxBlocks`" should be treated as the integer `3`. The module contains an object declaration part, two import statements referring to library modules, and one axiom.

```
include "../library-ontology"
include "../library"

numeric_symbol MaxBlocks = 3

module BW_SIMPLE;

  objects
```

```
    Table   : Supporter;
    B(1..MaxBlocks) : Thing;

  import TOWER;

  import NOCONCURRENCY;

  axioms
    Wide(Table);
```

7.1 The Library

7.1.1 Modules ACTOR, THEME

The first two modules are about actions in general. They introduce the concepts of an "actor" (performing agent) for an action and a "theme" (object affected) of an action. These concepts are necessary to express general principles such as

> Normally, to perform an action that affects an object x,
> the actor of the action has to be at the same place as x.

In order to express postulates about actions in general, we need to use the built-in sort, `action`, which is part of the enhanced version of MAD. All constants declared as actions automatically become objects of this sort.

We also need a sort that is not built in, to represent the actor or theme. For this, we add the following declaration to the library ontology:

```
sorts
  Thing;
```

The intended use for these modules is to be simply imported into action

descriptions in order to define actors and themes. Ideally, no renaming would be involved. Instead, the user will add axioms specifying any actor and theme for domain-specific actions.

An action may be performed by zero or more agents. An agent is not an actor for an action unless explicitly stated.

```
module ACTOR;

  fluents
    Actor(Thing,action): rigid;

  variables
    x: Thing;
    a: action;

  axioms
    default -Actor(x,a);
```

The theme of an action is a thing (an object) that the action affects. There may be zero or more themes for an action. A thing is not a theme for an action unless explicitly stated.

```
module THEME;

  fluents
    Theme(Thing,action): rigid;

  variables
    x: Thing;
    a: action;

  axioms
    default -Theme(x,a);
```

7.1.2 Modules ORDER, ASSIGN

These two modules are about an ordered relation among objects, and variables which may be assigned a value. They are abstract in the sense that they are not at all what is evoked by the phrase "commonsense reasoning". Rather, they are an attempt to capture pieces of underlying structure that is common to many commonsense relations and actions. Because of this, both of these modules are very different from modules ACTOR and THEME because they will almost always be imported with the constants renamed.

In order to define these relations, we need the following sort declarations:

```
sorts
  Domain;
  Range;
```

Here the words domain and range will be used to describe sets of objects that a relation may be about, or to describe fluents that may map an object from one set (the "domain") to an object from another set (the "range"). The use of Domain here is unrelated to the formal concept of the domain of a constant c, denoted $Dom(c)$, in a multi-valued signature (introduced in Section 3.1).

The Less relation is a transitive, irreflexive order relation among objects of sort Domain. There is no ordering relation among objects, unless explicitly stated otherwise.

```
module ORDER;

  fluents
    Less(Domain, Domain) : staticallyDetermined;

  variables
```

```
    s, s1, s2: Domain;

  axioms
    Less(s, s2) if Less(s, s1) & Less(s1, s2);
    default -Less(s,s1);
    constraint -Less(s,s);
```

According to module ASSIGN, each object in a given domain takes a value from a given range. The value remains the same in the absence of any action that assigns a new value to it.

```
module ASSIGN;

  actions
    Assign(Domain, Range);

  fluents
    Value(Domain)  : simple(Range);

  variables
    x: Domain;
    y: Range;

  axioms
    inertial Value(x);
    exogenous Assign(x,y);
    Assign(x,y) causes Value(x)=y;
```

Ideally, we would also add the following axiom to ASSIGN:

```
Theme(x, Assign(x,y));
```

Unfortunately, this would be useless with the current semantics of MAD, because

after an import where action `Assign` is renamed, it will become

$$\texttt{Theme(x, I}n\texttt{.Assign(x,y));}$$

but this does not state anything about the theme of the action in terms of which `Assign` is renamed. This is one of the shortcomings of the semantics of MAD, and in the conclusion to this dissertation (in Section 15.2.1) we list developing an improved semantics to solve this problem among topics for future work.

7.1.3 Module `MOVE`

In order to describe movement of things, such as pushing the box or walking (moving one's self), we need to express where things are located and have actions causing change in locations. Here we present a module about location and movement.

To represent possible locations, we have the following sort declaration:

```
sorts
  Place;
```

A thing has exactly one location at a time, which stays the same in the absence of actions. The effect of moving a thing is to cause it to be at a new place. Viewed like this, the location of a thing is a value that stays the same and action of moving assigns a new value to it. Therefore the description of moving imports module `ASSIGN`.

Lines beginning with `%` in MAD are comments.

```
module MOVE;

  actions
    Move(Thing,Place);
```

```
fluents
  Location(Thing): simple(Place);

variables
  x: Thing;
  p: Place;

import ASSIGN;
  Domain is Thing;
  Range is Place;
  Value(x) is Location(x);
  Assign(x,p) is Move(x,p);

axioms
  % Prevent trivial moves
  nonexecutable Move(x,p) if Location(x)=p;
```

7.1.4 Modules MOUNT, TOWER, TOP

In the classic Monkey and Bananas domain the monkey first walks to a new location and then needs to climb on the box before being able to grasp the bananas. Approaching the bananas in the horizontal plane can be accomplished by performing a single action—walking—which is a special case of moving. Approaching the bananas in the vertical direction is more of a challenge, because the monkey can't float in the air. As a general principle:

> Normally, it is impossible for an object not to be supported by anything.

For instance, the monkey is initially supported by the floor; after climbing the box, he is supported by the box. The box is always supported by the floor. If we adopt

the general understanding of "supported" as "held in the current position" then we can also say that the bananas are supported by the ceiling in the initial state, and by the monkey in the final state.

Here we present modules for support, for actions that change how things are supported, and the consequences of being supported.

The following declarations introduce the sort `Supporter`, which is a supersort of `Thing`:

```
sorts
  Supporter;

inclusions
  Thing << Supporter;
```

For example, the ground or the ceiling are not things (which must have a particular location) but they can still support things.

Fluent `Support(x)` indicates what thing `x` rests on (or what holds `x` in place). Objects are directly supported by exactly one supporter, and the action `Mount(x,s)` changes the supporter of `x` to become `s`. Both `x` and `s` are themes of this action. Again (as in module `MOVE`), this may be viewed simply as assigning a new value to a variable. Hence the description of mounting imports module `ASSIGN`.

Even though objects have exactly one direct supporter, many supporters may indirectly support them (by supporting their direct supporter). The indirect supporters of a thing are statically determined at each state by what its direct supporters are. A thing cannot be supported by itself, even indirectly. The irreflexive, transitive relation of indirect support is described by importing `ORDER`.

Since everything must be supported by a supporter and cycles in the supporting graph are prohibited, any action description that imports `MOUNT` must contain

a non-`Thing` object of sort `Supporter` in order to have possible models. (Without such special supporters it wouldn't be possible for all objects to be supported, yet none of them indirectly be supported by itself. Non-`Thing` supporters are like root nodes of a tree of supporting).

```
module MOUNT;

  actions
    Mount(Thing,Supporter);

  fluents
    Support(Thing):                  simple(Supporter);
    Supported(Supporter,Supporter): staticallyDetermined;

  variables
    x, y: Thing;
    s, s1: Supporter;

  import ASSIGN;
    Domain is Thing;
    Range is Supporter;
    Value(x) is Support(x);
    Assign(x,s) is Mount(x,s);

  import ORDER;
    Domain is Supporter;
    Less(s,s1) is Supported(s,s1);

  import THEME;

  axioms
    Theme(x,Mount(x,s));
    Theme(y,Mount(x,y));
```

```
  % Prevent trivial mounts
  nonexecutable Mount(x,s) if Support(x)=s;
  Supported(x,s) if Support(x)=s;
```

In its general form, module `MOUNT` allows many things to be held by the same supporter, such as books held by a shelf. However, sometimes we want to take the relative size of things and supporters into consideration and not allow multiple things to be supported by the same supporters. For example, when stacking blocks on top of each other. And in some cases we may want to allow both situations, depending on the kind of things/supporters involved, such as allowing a table to support multiple blocks, but blocks to support only a single other block.

The next module is designed for situations such as the blocks world, where we are building "towers" of things: each supporter may directly support only one thing, unless it is specified to be "wide" enough to support many. In addition, the supporter of a thing cannot be changed if it supports another thing. This prevents taking a stack of multiple blocks and mounting it somewhere else.

```
module TOWER;

  import MOUNT;

  fluents
    Wide(Supporter) : rigid;

  variables
    x, y : Thing;
    s    : Supporter;

  axioms
    % Two things cannot be directly on top of the same supporter
    % unless it's Wide
```

```
  constraint Support(x)=s & Support(y)=s & -Wide(s) -> x=y;
  % We can only mount things which are clear (not under others)
  nonexecutable Mount(x,s) if Support(y)=x;
  default -Wide(s);
```

When a thing is held up by another, it is usually at the same location as its supporter. For example, when the monkey is on the box, it is where the box is. Module `TOP` generalizes this consequence of being on a supporter to the case when the thing is located at a place related to the supporter, but not necessarily at the location of the supporter. (For example, if the supporter's dimensions are larger than the size of locations, it may extend out to a different location.)

Fluent `TopLocation(x)` is the location of the things whose support is `x`, if the supporter is a thing. By default, it's the same as the location of `x`.

```
module TOP;

  fluents
    Location(Thing):    simple(Place);
    Support(Thing):     simple(Supporter);
    TopLocation(Thing): staticallyDetermined(Place);

  variables
    x, y: Thing;
    p:    Place;

  axioms
    default TopLocation(x)=p if Location(x)=p;
    Location(x)=p if Support(x)=y & TopLocation(y)=p;
```

7.1.5 Modules NOCONCURRENCY, LOCAL

Modules ACTOR and THEME, presented earlier, were about actions in general, instead of specific actions, but they didn't express any effects of, or conditions on, the actions. Here, we present two modules expressing preconditions for actions in general.

The following module can be used if we want to formalize a domain in which actions cannot be executed concurrently. Here we use a variable for explicitAction because want to prevent co-occurrences of only actions which have not been renamed. (Renamed actions are implicitly declared and always co-occur with their renaming actions.)

```
module NOCONCURRENCY;

  variables
    a, a1: explicitAction;

  axioms
    nonexecutable a & a1 if a!=a1;
```

The following module states that an action can only be executed locally — all actors and themes must have the same location when the action is executed. For example, using this module together with MOUNT would require that, for Mount(x,s) to be executable, x and s should be at the same location. Such requirements for locality arise very often in commonsense reasoning. The monkey having to be next to the box before climbing on it is another example.

```
module LOCAL;

  import ACTOR;

  import THEME;
```

```
fluents
  Location(Thing): simple(Place);

variables
  x, y: Thing;
  a: action;

axioms
  nonexecutable a
    if (Actor(x,a) | Theme(x,a))
       & (Actor(y,a) | Theme(y,a))
       & Location(x)!=Location(y);
```

One issue to be careful about when using module `LOCAL` is to make sure that the actors/themes have been specified for exactly the same actions. If the actors/themes have been defined separately for some "equivalent" renamed/renaming actions, then they won't have the intended effect because the actions appear to be different and the locality axiom only considers a single action, not a pair of equivalent actions. This is a shortcoming of the semantics of MAD, and the list of topics for future work (presented in the conclusion to this dissertation in Section 15.2.1) includes developing an improved semantics to solve this problem.

7.2 Library Ontology

Here we show the part of the library ontology which includes the sorts and inclusions than what we have seen in this chapter. (The actual ontology file is larger because it includes sort and inclusion declarations for library modules we introduce in the coming chapters.)

```
% An ontology of sorts and their inclusion relations for our library

% Some "abstract sorts"
sorts
  Domain;
  Range;

% A Thing may be supported by another Thing or
% by a Supporter of a different sort, such as Ground
sorts
  Thing;
  Supporter;

inclusions
  Thing << Supporter;

sorts
  Place;
```

Figure 7.1 shows the sorts and subsort relations for the part of the library ontology seen so far. An arrow pointing from sort s_1 to s_2 indicates that s_2 is a subsort of s_1.

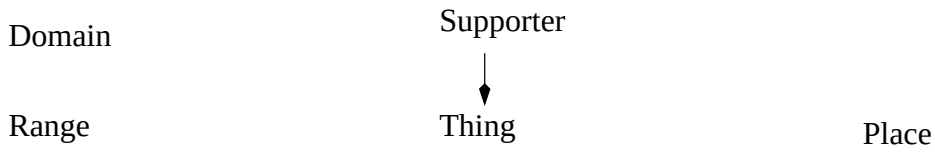

Figure 7.1: Sorts and subsort relations in the core library ontology

Chapter 8

Formalizing Domains with the Core Library

In this chapter we present MAD formalizations of a few classic domains from the knowledge representation literature: the Blocks World, Towers of Hanoi, and Monkey and Bananas. Along with each domain formalization, we present sample CCalc queries we ran for testing, and the results of these queries.

8.1 Blocks World

There are some blocks (of equal size) on a table. The blocks can be arranged into a set of towers.

The formalization in MAD is below, assuming there are 3 blocks.

The object declaration section here has two new features from the enhanced version of MAD: objects with arguments, and integer ranges. The declaration here tells us that `B(1)`, `B(2)`, `B(3)` are objects of sort `Thing`. The module `TOWER` postulates that only things not supporting other things may be mounted, and that two

things cannot be directly supported by the same supporter unless it is wide. There is a single domain-specific axiom in the blocks world that states that the `Table` is wide: it can support many blocks directly.

The first two lines below are instructions to include the library ontology and the library modules shown in the previous chapter.

```
include "../library-ontology"
include "../library"

numeric_symbol MaxBlocks = 3

module BW_SIMPLE;

  objects
    Table   : Supporter;
    B(1..MaxBlocks) : Thing;

  import TOWER;

  import NOCONCURRENCY;

  axioms
    Wide(Table);
```

Sample CCalc Query for Blocks World

We asked CCalc to start with blocks arranged into a tower, and find a plan (not longer than 10 steps) to build a tower in which the blocks are ordered differently.

```
:- query
maxstep :: 0..10;
0: support(b(1))=b(2),
   support(b(2))=b(3),
```

```
   support(b(3))=table;
maxstep: support(b(2))=table,
   support(b(3))=b(2),
   support(b(1))=b(3).
```

Here is the plan it found:

```
0:  mount(b(1), table)
1:  mount(b(2), table)
2:  mount(b(3), b(2))
3:  mount(b(1), b(3))
```

8.2 Towers of Hanoi

Another classic domain involving supported objects without movement between locations is the towers of Hanoi. There are three pegs and a set of disks with various sizes. The disks are placed on pegs to make towers, with the constraint that a disk can only have a smaller disk on top of it. Typically the disks are all on one peg initially and the goal is to move them all, one at a time, to one of the other pegs.

In our formalization we assume that disks on each peg are always in the correct order but we don't specify which disk is on which. We label disks with distinct positive integers, indicating their size.

The formalization is very similar to the blocks world.

The MAD formalization of this domain with three disks is shown below. Most of the knowledge is captured by importing library module `TOWER`. The only axiom expresses the domain-specific constraint that disks must be stacked in the order of their size. The operator "`<`" is only applicable when both sides of it are numerical.

```
include "../library-ontology"
include "../library"

numeric_symbol NumDisks = 3

module TOWERS_OF_HANOI;

  import TOWER;

  objects
    Peg(1..3):  Supporter;
    D(1..NumDisks) : Thing;

  variables
     i,j      : 1..NumDisks;

  import NOCONCURRENCY;

  axioms
    % A disk can only have a smaller disk on top of it
    constraint Support(D(i))=D(j) -> i < j;
```

Sample CCalc Query for Towers of Hanoi

We asked CCalc to start with all three disks on peg 1, and find a plan (not longer than 10 steps) to move all disks to peg 3.

```
:- query
maxstep :: 0..10;
0: support(d(1))=d(2),
   support(d(2))=d(3),
   support(d(3))=peg(1);
maxstep :
   support(d(1))=d(2),
```

```
   support(d(2))=d(3),
   support(d(3))=peg(3).
```

Here is the plan it found:

```
0:  mount(d(1), peg(3))
1:  mount(d(2), peg(2))
2:  mount(d(1), d(2))
3:  mount(d(3), peg(3))
4:  mount(d(1), peg(1))
5:  mount(d(2), d(3))
6:  mount(d(1), d(2))
```

8.3 Monkey and Bananas

There is a monkey in a room and there is a bunch of bananas hanging from the ceiling, which is too high for the monkey to reach. There is also a box. The monkey can walk to the box, push it under the bananas and climb on it to reach the bananas.

We describe this domain in three modules. First we describe what goes on at the floor level of the room, where the monkey can walk or push the box. Then we describe how the things in the domain are supported. Finally we give the full description which describes the vertical dimension of the domain.

```
include "../library-ontology"
include "../library"

module MBF;

% Monkey and Bananas: floor level

% An action may have a theme and may be executed by an agent.  To execute
% such an action, an agent has to be at the same place as the theme.
```

```
  import LOCAL;

  objects
    Monkey, Box: Thing;
    P1, P2, P3: Place;

  actions
    Walk(Place); PushBox(Place);

  variables
    x: Thing;
    p: Place;

% Walk is a special case of library action Move.

  import MOVE;
    Move(Monkey,p) is Walk(p);

% PushBox(p) has all the properties of library action Move(Monkey,p)

  import MOVE;
    Move(Monkey,p) is PushBox(p);

% PushBox(p) also has all the properties of library action Move(Box,p)

  import MOVE;
    Move(Box,p) is PushBox(p);

% Actions cannot be executed concurrently.

  import NOCONCURRENCY;

  axioms
```

```
    Actor(Monkey, PushBox(p));
    Theme(Box, PushBox(p));

module MBS;

% Monkey and Bananas: how things are supported

  import MBF;

  objects
    Bananas:        Thing;
    Floor, Ceiling: Supporter;

% TopLocation(x) is the location of the things whose support is x.

  import TOP;

  actions
    ClimbOn; ClimbOff; GetBananas;

  variables
    x: Thing;
    p: Place;
    s: Supporter;

% GetBananas is a special case of library action Mount.

  import MOUNT;
    Mount(Bananas, Monkey) is GetBananas;

% ClimbOn and ClimbOff are special cases of library action MOUNT.

  import MOUNT;
    Mount(Monkey,Box) is ClimbOn;
```

```
  import MOUNT;
    Mount(Monkey,Floor) is ClimbOff;

  axioms

    constraint Support(Box)=Floor;
    constraint Support(Monkey)=Floor | Support(Monkey)=Box;
```

Module MB below contains the full description of monkey and bananas, with the vertical dimension included. The library ontology declares sort Place, for movement in the horizontal dimension. The new sort Level, declared right before module MB, provides a vertical counterpart to Place, specific to this domain.

```
% This sort will be needed for the full description below
sorts
  Level;

module MB;

% Monkey and Bananas: full description

  import MBS;

  objects
    Lo, Hi: Level;

  fluents
    Elevation(Thing): simple(Level);
    TopLevel(Thing): staticallyDetermined(Level);

  variables
    x: Thing;
    f: Supporter;
```

```
    l: Level;

% To execute an action, an agent has to be at the same level as the theme.

  import LOCAL;
    Place is Level;
    Location(x) is Elevation(x);

% TopLevel(x) is the elevation of the things whose support is x.

  import TOP;
    Place is Level;
    Location(x) is Elevation(x);
    TopLocation(x) is TopLevel(x);

axioms

% Normally things are not tall: whatever is supported by x is at the same
% level as x.  (The default in module TOP.)  The box is an exception.
    TopLevel(Box)=Hi;

% Things directly supported by the floor are low and things directly
% supported by the ceiling are high.
    Elevation(x)=Lo if Support(x)=Floor;
    Elevation(x)=Hi if Support(x)=Ceiling;
```

Sample CCALC Query for Monkey and Bananas

We asked CCALC to start with the monkey, the bananas and the box all at different locations, and the bananas hanging from the ceiling, and asked it to find a plan (not longer than 10 steps) for the monkey to get the bananas.

```
:- query
maxstep :: 1..10;
```

```
0: location(monkey)=p1,
   location(bananas)=p2,
   support(bananas)=ceiling,
   location(box)=p3;
maxstep: support(bananas)=monkey.
```

Here's the plan it found:

```
0:  walk(p3)
1:  pushbox(p2)
2:  climbon
3:  getbananas
```

Chapter 9

Extending the Library: Module CARRIER

9.1 Introduction

In the preceding chapter we showed how the simple library from Chapter 7 could be used to formalize classic domains from the literature.

In this chapter, we extend the library by adding a module about carriers — containers that can change their locations along with their contents, or vehicles that move around along with their passengers and luggage. Then we use the library to formalize several action domains familiar from the literature on commonsense reasoning and planning. One of these examples is the briefcase that Ed Pednault used twenty years ago to carry a book to his office [Pednault, 1988]. Long before that, a boat was used by missionaries and cannibals to cross the river [Amarel, 1968]. More recently, passengers took flights on planes to get to their destinations [Gelfond, 2006]. At first glance these domains seem quite different, but when examined carefully a

common feature shows up: they all have to do with carriers.

Using the library to formalize these domains not only is more natural, leading to more concise action descriptions, but it also makes it easier for us to recognize structural similarities between action domains.

In the next section we show the general-purpose library module `CARRIER`. This is followed by several sections formalizing diverse domains, illustrating how using the library simplifies them greatly.

9.2 A New Library Module: `CARRIER`

The library module `CARRIER` contains the core knowledge common to the action domains we formalize in this chapter. (A detailed explanation of the module follows it.)

We add the following declarations to the library ontology for use in the module `CARRIER`.

```
sorts
  Carrier;
  Person;
  Vehicle;

inclusions
  Carrier << Thing;
  Person << Carrier;
  Vehicle << Carrier;
```

Sorts `Person` and `Vehicle` are described as subsorts of `Carrier`,and consequently subsorts of `Thing`. (A person is a carrier because he can carry things in his hands or pockets. This fact will become essential in some examples.) Sort `Supporter` is a supersort of `Thing`, which makes all carriers also supporters.

As a result of adding these declarations, the sort and subsort relations in the library ontology now become as shown in Figure 9.1.

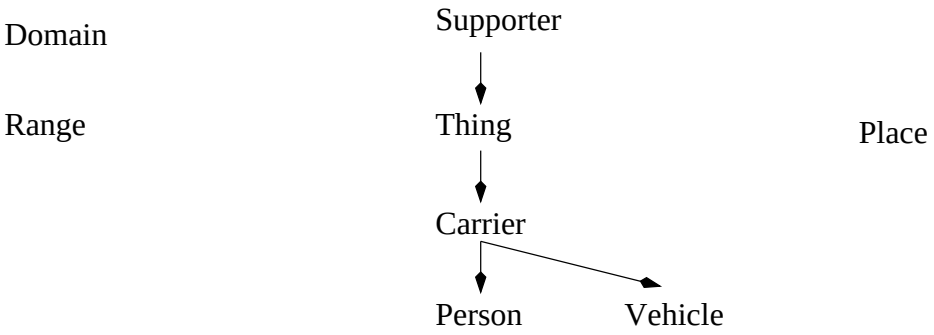

Figure 9.1: Sorts and subsort relations in the ontology after adding module CARRIER

The library module CARRIER is shown below (A detailed explanation of the module follows it.)

```
module CARRIER;

  objects
    Ground : Supporter;

  actions
    Load(Thing, Carrier);
    Unload(Thing);

  fluents
    Big(Carrier),
    DriverRequired(Vehicle)  : rigid;
    TooSmallToSupport(Carrier, Thing),
    Holds(Carrier, Thing)    : staticallyDetermined;

  variables
    x,y  : Thing;
    c    : Carrier;
```

```
  m,m1 : Person;
  v,v1 : Vehicle;
  p    : Place;

import MOUNT;
  Mount(x,c) is Load(x,c);

import MOUNT;
  Mount(x,Ground) is Unload(x);

import ORDER;
  Domain is Thing;
  Less(c,x) is TooSmallToSupport(c,x);

import MOVE;

import TOP;

import LOCAL;

axioms
  constraint Support(x)=y -> Carrier(y);

  default Big(c);
  constraint Support(m)=m1 -> -Big(m);
  constraint Support(v)=v1 -> -Big(v);

  constraint Support(x)=c -> -TooSmallToSupport(c,x);
  TooSmallToSupport(m,v);

  nonexecutable Move(v,p) if DriverRequired(v)
                           & -exists m Support(m)=v;
  default DriverRequired(v);
```

```
Holds(c,x) if Supported(x,c);
default -Holds(c,x);
```

Module CARRIER contains six import statements, referring to five other library modules: MOUNT, ORDER, TOP, LOCAL, and MOVE. Recall that, very briefly, these modules can be described as follows:

- MOVE: Introduces a fluent Location(x) (with domain Place) for every object x of sort Thing and describes action Move for moving a thing to a location.
- MOUNT: Introduces a fluent Support(x) (with domain Supporter) for every object x of sort Thing and postulates that every object of sort Thing must be (possibly indirectly) supported by an object of sort Supporter. The action Mount changes the support of a thing.
- ORDER: Formalizes a transitive, irreflexive order relation Less on objects of sort Domain.
- TOP: Postulates that every object supported by another is at the same location as its supporter, by default.
- LOCAL: Postulates that the parties to an action must all be at the same location.

The CARRIER module declares a single object Ground, of sort Supporter. According to module MOUNT, every Thing must be supported by a Supporter. The object Ground provides such a domain-independent Supporter.

The two imports of library module MOUNT serve to describe two actions associated with carriers, Load and Unload, by renaming action Mount. As a result, the effects of loading a thing onto a carrier and unloading a thing are reflected in the support of the thing. The second instance shows that Unload is a special case of Mount where the support becomes Ground.

The import of ORDER expresses that the newly introduced fluent TooSmallToSupport represents a transitive, irreflexive order relation on objects of sort Thing.

The import of MOVE adds knowledge about moving things. Adding TOP reflects the fact that a thing supported by a carrier is at the same location as the carrier. Adding LOCAL (in conjunction with some axioms imported in module MOUNT—those specifying themes of the action) restricts the action Load(x,c) to cases where the thing x is at the same location as the carrier c.

The first axiom restricts the supporting things to be carriers.

The three axioms involving "Big" state that, by default, persons and vehicles are too big to be supported by objects of the same sort.

The two axioms involving TooSmallToSupport state that a thing can only be supported by a carrier that is not too small to support it, and that persons are always too small to support vehicles.

The two axioms involving DriverRequired state that the action of moving a vehicle v to a place p is not possible if a driver is required for v and there are no persons which are supported by v. By default, every vehicle requires a driver. This is what sets vehicles apart from other carriers.

The last two axioms define statically determined fluent Holds, whose value is always determined by the value of Support, in order to more conveniently talk about which carriers hold what things.

9.3 Pednault's Briefcase Domain

The following description of the briefcase domain is from [Pednault, 1988]:

> Suppose that we have a world that consists of three objects—a briefcase, a dictionary, and a paycheck—each of which may be situated in one of

two locations: the home or the office. Actions are available for putting objects in the briefcase, and for taking objects out, as well as for carrying the briefcase between the two locations. Initially, the briefcase, the dictionary, and the paycheck are at home; the paycheck is in the briefcase, but the dictionary is not. The goal is to have the briefcase and dictionary at the office and the paycheck at home.

We may represent this domain using five fluent constants. Three of them describe the locations of the briefcase, the dictionary, and the paycheck; the possible locations are the home and the office. The other two indicate whether the dictionary or the paycheck are in the briefcase. Out of the 32 combinations of values of these fluents, only 18 represent possible states of the world, because when the paycheck is in the briefcase, both have to be at the same place, and similarly for the dictionary.

A description of the briefcase domain in MAD is shown below.

```
include "../library-ontology"
include "../library"

sorts
  Item;

inclusions
  Item << Thing;

module BRIEFCASE;

  objects
    Paycheck, Dictionary : Item;
    Briefcase            : Carrier;
    Home, Office         : Place;
```

```
actions
  PutIn(Item);
  TakeOut(Item);
  MoveB(Place);

variables
  i : Item;
  p : Place;

import CARRIER;
  Load(i,Briefcase) is PutIn(i);
  Unload(i) is TakeOut(i);
  Move(Briefcase,p) is MoveB(p);

import NOCONCURRENCY;
```

The description declares `Item` to be a sort, and it postulates that `Item` is a subsort of the sort `Thing`.

Apart from the declarations at the beginning of module `BRIEFCASE`, all the core knowledge necessary for the briefcase domain comes from importing the library module `CARRIER`, which was described above, in Section 9.2. Therefore no axioms are needed.

The import statement from module `BRIEFCASE`

```
import CARRIER;
  Load(i,Briefcase) is PutIn(i);
  Unload(i) is TakeOut(i);
  Move(Briefcase,p) is MoveB(p);
```

tells us that the action `PutIn(i)` has all properties that are postulated for the action `Load(x,c)` in the library module `CARRIER` when the thing `x` is an item and

the carrier `c` is `Briefcase`, and similarly for the actions `TakeOut(i)` and `MoveB(p)`. For example, with this import the axiom

```
Move(x,p) causes Location(x)=p;
```

from module `CARRIER`[1] (where `x` is a variable for things) has the same effect as if we had written the axiom

```
MoveB(p) causes Location(Briefcase)=p;
```

in module `BRIEFCASE`.

One other assumption about the fluent `Location` in the module `CARRIER` is that it satisfies the commonsense law of inertia—the location of a thing is presumed to remain unchanged in the absence of information to the contrary. Furthermore, it is impossible to move a thing to its current location. These assumptions, just as the assumption about the effect of `Move(x,p)` on `Location(x)`, are "inherited" by `BRIEFCASE` from `CARRIER`. In the absence of a library of standard action descriptions, many such axioms would have to be explicitly included in module `BRIEFCASE`.

The fact that a carrier `c` is holding a thing `x` is described in module `CARRIER` by the truth-valued fluent `Holds(c,x)`.[2] Executing action `Load(x,c)` makes this fluent true, and executing `Unload(x)` makes it false.

According to the axioms of `CARRIER`, the action `Load(x,c)` is nonexecutable if `Location(x)` is different from `Location(c)`. For instance, the action of putting the dictionary in the briefcase cannot be executed when the dictionary is at home and the briefcase is at the office.

[1] To be precise, this axiom is found in the library module `MOVE`, which is imported by `CARRIER`.
[2] Not to be confused with the use of the relation *Holds* in the situation calculus.

Sample CCalc Query for the Briefcase Domain

We asked CCalc to solve the briefcase planning problem:

```
:- query
maxstep :: 1..10;
0: location(briefcase)=home,
   holds(briefcase,paycheck),
   location(dictionary)=home,
   -holds(briefcase,dictionary);
maxstep:
   location(briefcase)=office,
   location(dictionary)=office,
   location(paycheck)=home.
```

It determined that the shortest plan consists of 3 actions:

```
0:  takeout(paycheck)
1:  putin(dictionary)
2:  moveb(office)
```

In a different test, we instructed CCalc to display the list of all states and all transitions of the transition system represented by the formalization of the briefcase domain, and it found 18 states and 60 transitions, as we had expected.

9.4 The Dictionary and Paycheck Disguised as Humans

In this section we formalize two commonsense domains having to do with humans and vehicles, which are structurally very similar to the briefcase domain discussed above. One is a simplified version of the familiar missionaries and cannibals puzzle [Amarel, 1968], in which there are no cannibals—just three persons who want to cross the river, and a boat that holds two. The second, inspired by [Gelfond, 2006], involves travel by air.

Amarel [1968] discusses how the missionaries and cannibals domain may be represented in different ways, and points out that representing missionaries and cannibals as named individuals will lead to having more states than a representation that simply represents the numbers of missionaries and cannibals as two groups. In this section we present a version with individual missionaries as objects. In Section 11.4 we present an alternative formalization where we consider just the number of missionaries on each bank of the river and in the boat.

In the modules `MISSIONARIES` and `AIRTRAVEL` below, sort `Person` is used in place of sort `Item` from module `BRIEFCASE` above. However, there is no need for sort declarations here, because `Person` and `Vehicle` are described in the library ontology.

```
include "../library-ontology"
include "../library"

module MISSIONARIES;

  objects
    Miss(1..3) : Person;
    Boat : Vehicle;
    Bank1, Bank2 : Place;

  actions
    Board(Person);
    Disembark(Person);
    CrossTo(Place);

  variables
    m : Person;
    p : Place;
```

```
  import CARRIER;
    Load(m,Boat) is Board(m);
    Unload(m) is Disembark(m);
    Move(Boat,p) is CrossTo(p);

  import NOCONCURRENCY;

  axioms

    % The boat can carry at most two (i.e. not all three)
    constraint -forall m Holds(Boat,m);
```

According to the library module CARRIER, persons are too small to carry vehicles. So a missionary cannot hold the boat on his back.

```
include "../library-ontology"
include "../library"

module AIRTRAVEL;

  objects
    George, Laura   : Person;
    AirForce1       : Vehicle;
    Austin, Lubbock : Place;

  actions
    Board(Person);
    Disembark(Person);
    Fly(Place);

  variables
    m: Person;
    p : Place;
```

```
  import CARRIER;
    Load(m,AirForce1) is Board(m);
    Unload(m) is Disembark(m);
    Move(AirForce1,p) is Fly(p);

  import NOCONCURRENCY;

  axioms
    % the pilot is disregarded in this formalization
    -DriverRequired(AirForce1);
```

Unlike `BRIEFCASE`, each of the modules `MISSIONARIES` and `AIRTRAVEL` includes an axiom section, to describe the domain-specific assumptions that are not covered by the axioms in the imported modules. In `MISSIONARIES`, the only domain-specific assumption is that the boat holds two. In `AIRTRAVEL`, we postulate that `AirForce1` is an exception to the above-mentioned default about vehicles (not because it is fully automatic, of course, but because our simplified formalization disregards the presence of a pilot).

Sample CCALC Query for the Missionaries Domain

We asked CCALC to solve the missionaries planning problem, where all missionaries are on `Bank1` initially and they are all on `Bank2` at the end:

```
:- query
maxstep :: 8..9;
0: location(miss(1))=bank1,
   location(miss(2))=bank1,
   location(miss(3))=bank1,
   -holds(boat,miss(1)),
   -holds(boat,miss(2)),
   -holds(boat,miss(3));
```

```
maxstep:
   location(miss(1))=bank2,
   location(miss(2))=bank2,
   location(miss(3))=bank2,
   -holds(boat,miss(1)),
   -holds(boat,miss(2)),
   -holds(boat,miss(3)).
```

It determined that the shortest plan consists of 9 actions:

```
0:  board(miss(1))
1:  board(miss(2))
2:  crossto(bank2)
3:  disembark(miss(2))
4:  crossto(bank1)
5:  board(miss(3))
6:  crossto(bank2)
7:  disembark(miss(3))
8:  disembark(miss(1))
```

Sample CCalc Query for the Airtravel Domain

We asked CCalc to find a plan for the following query. Initially, `Laura` and `AirForce1` are in `Austin` with `George` supported by `AirForce1`. We wish to have `Laura` and `AirForce1` in `Lubbock`, but want to keep `George` in `Austin`.

```
:- query
maxstep :: 1..10;
0: location(laura)=austin,
   location(airforce1)=austin,
   support(airforce1)=ground,
   holds(airforce1,george),
   -holds(airforce1,laura);
maxstep:
```

```
location(george)=austin,
location(laura)=lubbock,
location(airforce1)=lubbock,
-holds(airforce1,laura).
```

It determined that the shortest plan consists of 4 actions:

```
0:  disembark(george)
1:  board(laura)
2:  fly(lubbock)
3:  disembark(laura)
```

9.5 Takeoff and Landing

Module `AIRTRAVEL_AIR` is an enhancement of the air travel example that takes into account the need to take off before flying anywhere and to land after that. It imports module `AIRTRAVEL` and declares two additional actions, `TakeOff` and `Land`.

The effects of these actions are described here using the fluent `Support(x)`, declared in the library module `MOUNT`. Executing action `TakeOff` changes the value of `Support(AirForce1)` to `Air`; after executing action `Land`, its value becomes `Ground`. Both `Ground` and `Air` are objects of sort `Supporter`. `Ground` is declared in the library; `Air` is specific for the module `AIRTRAVEL_AIR`.

The file "`airtravel`" included in the first line below contains the module `AIRTRAVEL` shown in the preceding section.

```
include "airtravel"

module AIRTRAVEL_AIR;

  import AIRTRAVEL;
```

```
objects
  Air : Supporter;

actions
  TakeOff; Land;

variables
  x : Thing;
  m : Person;
  p : Place;

import MOUNT;
  Mount(AirForce1, Air) is TakeOff;

import MOUNT;
  Mount(AirForce1, Ground) is Land;

axioms
  % Must take off before flying
  nonexecutable Fly(p) if Support(AirForce1)=Ground;
   % Must land before getting in or out
  nonexecutable (Board(m) | Disembark(m))
                if Support(AirForce1)!=Ground;
  % Only the plane can be freely flying
  constraint Support(x)=Air -> x=AirForce1;
```

Sample CCalc Query for the Enhanced Airtravel Domain

We asked CCalc to find a plan for the same query. The length of the shortest plan increased from 4 to 6 steps:

```
0:  board(laura)
1:  disembark(george)
2:  takeoff
```

```
3:  fly(lubbock)
4:  land
5:  disembark(laura)
```

9.6 Pednault's Briefcase Revisited

The enhacement of Pednault's example shown below takes into account the fact that the briefcase doesn't move to the office by itself; the owner carries it with him. We assume here that he walks to the office.

```
include "../library-ontology"
include "../library"

sorts
  Item;

inclusions
  Item << Thing;

module BRIEFCASE_ED;

  objects
    Ed                    : Person;
    Paycheck, Dictionary  : Item;
    Briefcase             : Carrier;
    Home, Office          : Place;

  actions
    PutIn(Item);
    PickUp(Thing);
    PutDown(Thing);
    Walk(Place);
```

```
variables
  i : Item;
  x : Thing;
  p : Place;

import CARRIER;
  Load(x,Ed) is PickUp(x);
  Unload(x) is PutDown(x);
  Move(Ed,p) is Walk(p);

import CARRIER;
  Load(i,Briefcase) is PutIn(i);
  Unload(x) is false;
  Move(x,p) is false;

import NOCONCURRENCY;

axioms
  TooSmallToSupport(Briefcase,Ed);
  nonexecutable PutDown(x) if -Holds(Ed,x);
```

Module `CARRIER` is imported here twice: first to describe the new actions of picking up and putting down items, and then, as in `BRIEFCASE`, to describe putting an item in the briefcase. (The action of taking an item out of the briefcase is no longer necessary in the presence of the new action `PickUp`.) The action `MoveB` from the simpler formalization is not available anymore. Instead, `Walk` is declared to be an action that changes Ed's location, and consequently the locations of all things that Ed carries.

The enhanced formalization of the briefcase domain has two axioms. The first of them uses the relation `TooSmallToSupport` between two things, which is introduced in module `CARRIER` for the purpose of specifying when a carrier is "too

small" to hold a thing. This relation is postulated to be false by default, and we have already seen one exception to this default: humans are too small to hold vehicles. Now we postulate also that Ed Pednault's briefcase is too small to enclose its owner.

The second axiom says that Ed can put down a thing only if he is holding it.

Sample CCalc Query for the Briefcase Domain with Ed

We asked CCalc to solve the briefcase planning problem:

```
:- query
maxstep :: 1..10;
0: location(briefcase)=home,
   location(dictionary)=home,
   location(ed)=home,
   holds(briefcase,paycheck),
   -holds(briefcase,dictionary),
   -holds(ed,dictionary),
   -holds(ed,briefcase);
maxstep:
   location(briefcase)=office,
   location(dictionary)=office,
   location(paycheck)=home.
```

It determined that the shortest plan consists of 4 actions:

```
0:  putin(dictionary)
1:  pickup(briefcase)
2:  putdown(paycheck)
3:  walk(office)
```

There is another alternative to the plan above: instead of putting the dictionary in the briefcase, he can simply pick it up and carry it in his other hand.

Chapter 10

Extending the Library: Module MOVE_IN_REGION

10.1 A New Library Module: MOVE_IN_REGION

Sometimes the concept of location is not enough by itself and we need to talk about groups of locations. For example, a house consists of many rooms. A room, in turn, may have multiple locations where objects may be. The library module `MOVE` introduced in Section 7.1.3 only dealt with single locations (represented by ontology sort `Place`). In this chapter we extend that module by introducing a new library module for representing movement within regions: `MOVE_IN_REGION`.

In order to represent such situations, we included the following declarations in the library ontology, to introduce "regions" which are a supersort of places.

```
sorts
  Region;

inclusions
```

```
Place << Region;
```

A `Region` is a (potentially) larger kind of `Place`. It may contain one or more `Place`s or other `Region`s.

As a result of adding these declarations, the sort and subsort relations in the library ontology now become as shown in Figure 10.1.

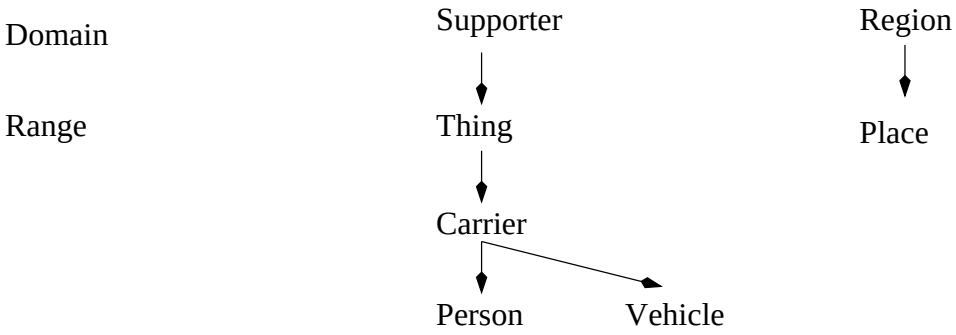

Figure 10.1: Sorts and subsort relations in ontology after adding `MOVE_IN_REGION`

The module `MOVE_IN_REGION` is an extension of `MOVE` and `ORDER`. The notion of a place being a part of a region, and, by transitivity, being a part of any regions encompassing that region, is captured by importing `ORDER`.[1] This module restricts movement: for a thing to be moved from one place to another, both places must be part of a common region, and the region must be small enough to be "movable" across.[2] The intended usage of this module would redefine both the action `Move` and the fluent `Movable` during importing, to indicate that the region is "movable across by the action which renames `Move`." The example in the next section demonstrates this usage.

```
module MOVE_IN_REGION;
```

[1] This module is inspired by John McCarthy's classic *Advice Taker* paper [McCarthy, 1959]. Our choice of the word "At" to denote places and regions being part of other regions follows that paper.

[2] Again, the term "Movable" is inspired by [McCarthy, 1959]. It is a generalization of the two terms used in that paper: "Walkable" and "Drivable."

```
import MOVE;

fluents
  At(Region, Region) : staticallyDetermined;
  Movable(Region): rigid;

variables
  x: Thing;
  p,p1: Place;
  r, r1: Region;

import ORDER;
  Domain is Region;
  Less(r,r1) is At(r,r1);

axioms
  nonexecutable Move(x,p) if Location(x)=p1
                           & -exists r (At(p1,r) & At(p,r)
                                        & Movable(r));
  default -Movable(r);
```

10.2 The Oldest Planning problem in AI: Getting to the Airport

John McCarthy [1959] explained the fact that he needed a car to get to the airport by noting that his home and the airport do not belong to a sufficiently small, "walkable", region. They are in the same county, and counties are "drivable"—small enough to drive across. He could get to the airport by first walking to the car, which is at his home also (this is possible because his home is walkable) and then driving his car to the airport.

In the library module MOVE_IN_REGION we generalized this by introducing the concept of a "movable" region and postulating that the action Move(x,p) is nonexecutable unless place p lies within a movable region that contains Location(x). In the formalization below, we formalize McCarthy's example using MOVE_IN_REGION, along with library module CARRIER from the preceding chapter.

```
include "../library-ontology"
include "../library"

module AIRPORT;

  objects
    John                  : Person;
    Car                   : Vehicle;
    Desk, Garage, Airport : Place;
    Home, County          : Region;

  actions
    Walk(Place);
    Drive(Place);
    Board;
    Disembark;

  fluents
    Walkable(Region),
    Drivable(Region) : rigid;

  variables
    p : Place;
    r : Region;

  import CARRIER;
    Load(John,Car) is Board;
```

```
    Unload(John) is Disembark;
    Move(Car,p) is Drive(p);

  import MOVE_IN_REGION;
    Move(Car,p) is Drive(p);
    Movable(r) is Drivable(r);

  import MOVE_IN_REGION;
    Move(John,p) is Walk(p);
    Movable(r) is Walkable(r);

  import NOCONCURRENCY;

  axioms

    constraint Location(Car)!=Desk;

    % Our "geography":
    At(Desk, Home);
    At(Garage, Home);
    At(Home, County);
    At(Airport, County);

    Walkable(Home);
    Drivable(County);
```

Sample CCalc Query for Getting to the Airport

We asked CCalc to solve the classic planning query: John is at his desk, the car is in the garage. How can he get to the airport?

```
:- query
maxstep :: 0..3;
0: location(john)=desk,
```

```
   location(car)=garage;
maxstep: location(john)=airport.
```

It determined that the shortest plan consists of 3 actions:

```
0:  walk(garage)
1:  board
2:  drive(airport)
```

10.3 The Logistics Domain

The logistics domain, introduced in [Veloso, 1992], is described as follows[3]:

> There are several cities, each containing several locations, some of which are airports. There are also trucks, which can drive within a single city, and airplanes, which can fly between airports. The goal is to get some packages from various locations to various new locations.

A MAD formalization of this domain is shown below. The condition that trucks can only drive within a single city is similar to the limitations on walking and driving in McCarthy's example, and it is expressed here by importing module `MOVE_IN_REGION`.

```
include "../library-ontology"
include "../library"

sorts
  City; Airport;
  Truck; Airplane;
  Package;
```

[3]Taken from the webpage of the first International Planning Competition, `ftp://ftp.cs.yale.edu/pub/mcdermott/ aipscomp-results.html`

```
inclusions
  City << Region;
  Airport << Place;
  Truck << Vehicle;
  Airplane << Vehicle;
  Package << Thing;

module LOGISTICS;

  actions
    Go(Vehicle, Place);

  fluents
    Drivable(City)     : rigid;

  variables
    p : Place;
    c : City;
    v : Vehicle;
    t : Truck;
    a : Airplane;

  import CARRIER;
    Move(v,p) is Go(v,p);

  import MOVE_IN_REGION;
    Move(t,p) is Go(t,p);
    Movable(c) is Drivable(c);

  import NOCONCURRENCY;

  axioms
    -DriverRequired(v);
```

```
  constraint Location(a)=p -> Airport(p);
  Drivable(c);
```

This representation of logistics is abstract, in the sense that it does not declare objects corresponding to specific vehicles, places and packages. A module describing a concrete logistics domain would import module `LOGISTICS`, declare its objects, and provide axioms describing the `At` relation between places and cities. For example,

```
module LOGISTICS_SPECIFIC;

  import LOGISTICS;

  objects
    T1, T2                      : Truck;
    A1                          : Airplane;
    Pack1, Pack2, Pack3, Pack4  : Package;
    L1                          : Place;
    L2, L3                      : Airport;
    C1, C2                      : City;

  axioms
    At(L1, C1);
    At(L2, C1);
    At(L3, C2);
```

Sample CCalc Query for the Logistics Domain

We asked CCalc for a plan to solve the following planning problem: All the packages are at a certain place, on the ground. The vehicles are at another place, an airport, in the same city. How can the packages be transported to an airport in another city?

```
:- query
maxstep :: 10..11;
0: location(pack1)=l1,
   location(pack2)=l1,
   location(pack3)=l1,
   location(pack4)=l1,
   location(t1)=l2,
   location(t2)=l2,
   location(a1)=l2;
maxstep:
   location(pack1)=l3,
   location(pack2)=l3,
   location(pack3)=l3,
   location(pack4)=l3.
```

It determined that the shortest plan consists of 11 actions:

```
0:  go(t2, l1)
1:  load(pack1, t2)
2:  load(pack2, t2)
3:  load(pack4, t2)
4:  load(pack3, t2)
5:  go(t2, l2)
6:  load(pack3, a1)
7:  load(pack1, a1)
8:  load(pack2, a1)
9:  load(pack4, a1)
10:  go(a1, l3)
```

Chapter 11

Extending the Library: Modules TIME, TRANSFER and BUY

We saw in Chapter 6 that the extended version of MAD treats integers as built-in objects and can do comparisons on numerical objects. When we have integers available, it is possible to enhance our action domains in many ways.

Here we present three library enhancements related to integers: one is about the notion of time, and actions with variable durations; the other two are about numeric-valued resources and their transfer. They treat the notion of transferring resources both generally and also in the context of buying and selling.

11.1 A New Library Module: TIME

Library module `TIME` declares the integer-valued fluent `Time` to keep track of how much "time" has gone by in a domain. Unlike most of the fluents we've encountered so far, `Time` is not inertial. Instead, each action has a duration, which is 1 by default, and each transition increases the value of `Time` by the duration of the

actions executed in that transition. All actions that are concurrently executed need to have the same duration.

There is also a special action of "waiting" for a certain amount of time. This allows time to move forward even in the absence of any "real" action.

An important feature of this module is that it uses a numeric symbol named `MaxTime` instead of a fixed integer. The value of this symbol is set in a file to be included before the library. This way, the user of the library module may set a `MaxTime` value of his choosing.

```
module TIME;

  actions
    Wait(1..MaxTime);

  fluents
    Time : simple(0..MaxTime);
    Duration(action) : rigid(1..MaxTime);

  variables
    t_s   : 1..MaxTime;
    t     : 0..MaxTime;
    a     : action;
    a_exp : explicitAction;

  axioms
    a_exp causes Time=t if t=Time+Duration(a_exp);
    default Duration(a)=1;
    exogenous Wait(t_s);
    Duration(Wait(t_s))=t_s;
```

An interesting result of having time is that actions which might have been called trivial otherwise now become nontrivial. For example, even if an action has

no effect in the traditional sense, as a result of executing it time will pass, and transitions in the transition system will always go to a new state.

11.2 Briefcase with Time and Duration

We may enhance the briefcase domain from Section 9.3 to add time, and conditions dependent on time. The enhancement shown in module `BRIEFCASE_TIME` for instance, expresses that Pednault's office opens at a certain time, and that the action of going to the office cannot be executed before then.

The file "`briefcase`" included in the first line below contains the module `BRIEFCASE` shown in Section 9.3.

```
include "briefcase"

module BRIEFCASE_TIME;

  import BRIEFCASE;

  import TIME;

  axioms
    nonexecutable MoveB(Office) if Time<4;
```

Sample CCalc Query for Briefcase with Time

We asked CCalc to solve the briefcase planning problem, assuming that initially `Time=0`.

```
:- query
maxstep :: 1..10;
0: location(briefcase)=home,
   holds(briefcase,paycheck),
```

```
   location(dictionary)=home,
   -holds(briefcase,dictionary),
   time=0;
maxstep:
   location(briefcase)=office,
   location(dictionary)=office,
   location(paycheck)=home.
```

In this formalization, Pednault's planning problem can be solved in 4 steps instead of 3. Before executing `MoveB(Office)`, besides taking out the paycheck and putting in the briefcase, an action of waiting must be executed, waiting for enough time units so that `Time` ≥ 4.

```
0:  putin(dictionary)
1:  wait(6)
2:  takeout(paycheck)
3:  moveb(office)
```

11.3 A New Library Module: TRANSFER

Another phenomenon that becomes a possibility when we have integer-valued constants and arithmetic is domains where certain countable/measurable resources are transferred around. For example, in the missionaries and cannibals domain, instead of reasoning about specific individuals, we can reason about the number of missionaries and cannibals on one of the banks or in the boat.

For this, we add the following declarations to our library ontology:

```
sorts
  Resource;
  Accumulator;

inclusions
```

```
Accumulator << Thing;
```

As a result of adding these declarations, the sort and subsort relations in the library ontology now become as shown in Figure 11.1.

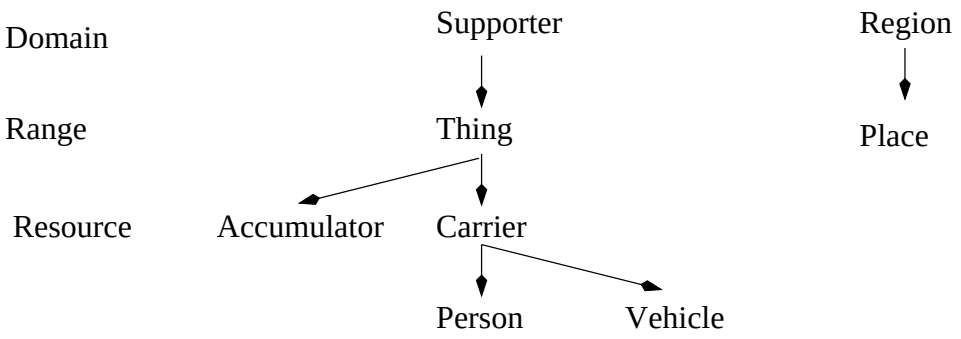

Figure 11.1: Sorts and subsort relations in ontology after adding module TRANSFER

Library module TRANSFER, presented below, formalizes the notion of resource "accumulators" holding certain amounts of resources.

```
module TRANSFER;

  actions
    Transfer(1..MaxAmount, Resource, Accumulator, Accumulator);

  fluents
    Amount(Resource, Accumulator) : simple(0..MaxAmount);

  variables
    m,m1 : 0..MaxAmount;
    n    : 1..MaxAmount;
    r    : Resource;
    x,y  : Accumulator;
    p    : Place;

  import LOCAL;
```

```
axioms

  exogenous Transfer(n,r,x,y);

  inertial Amount(r,x);

  nonexecutable Transfer(n,r,x,y) if Amount(r,x) < n;
  nonexecutable Transfer(n,r,x,y) if Amount(r,y)=m1
                                  & MaxAmount < m1+n;

  Transfer(n,r,x,y) causes Amount(r,x)=m if Amount(r,x)=m1 & m1=m+n;
  Transfer(n,r,x,y) causes Amount(r,y)=m1 if Amount(r,y)=m & m1=m+n;

  % These will help us say that transfers obey locality
  Theme(x, Transfer(n,r,x,y));
  Theme(y, Transfer(n,r,x,y));
```

Numeric-valued fluent `Amount` represents how much of a certain resource is held by an accumulator, and the action `Transfer` is used to represent certain amounts of a resource being moved from one accumulator to another.

As in module `TIME`, a numeric symbol (instead of a fixed integer) is used to specify the upper boundary of an integer range in the library. Here it is `MaxAmount`.

The axioms specify the preconditions and effects of `TRANSFER`.

This module has an important limitation due to the MAD language. It requires that at most a single `Transfer` action be applied to each resource at a time. If there are concurrent `Transfer` actions that act on the same resource, then the resulting amount will be incorrect, because the causal laws specifying the effect of the action are written for single actions. This limitation is due to the MAD language not containing any kind of "additive fluents" [Lee and Lifschitz, 2003]—

fluents that can be used to correctly calculate the aggregated effects of concurrent actions on numeric-valued fluents. Enhancing MAD with such fluents is one of those we list as future work in Section 15.2.1.

11.4 Missionaries as Resources

In the formalization of the missionaries domain presented in Section 9.4, individual missionaries were all objects. In this variant, we formalize all the missionaries as a single object — a resource, of which different numbers may be accumulated by the two banks of the river, or the boat. The actions of boarding and disembarking now become instances of transferring resources.

The formalization below uses two domain-specific numeric symbols: `MaxMiss` indicates the number of missionaries, and `MaxBoatCapacity` indicates how many missionaries can fit in the boat at a time.

```
include "../library-ontology"
include "../library"

sorts
   RiverBank;

inclusions
   RiverBank << Accumulator;

% Maximum number of missionaries
numeric_symbol MaxMiss=9

% Maximum capacity of the boat
numeric_symbol MaxBoatCapacity=5

module MISSIONARIES;
```

```
objects
  M : Resource;
  P1, P2 : Place;
  Bank1, Bank2 : RiverBank;
  Boat : Accumulator;

actions
  Board(1..MaxMiss, RiverBank);
  Disembark(1..MaxMiss, RiverBank);
  CrossTo(Place);

variables
  n : 1..MaxMiss;
  b : RiverBank;
  p : Place;

import TRANSFER;
  Transfer(n,M,b,Boat) is Board(n,b);

import TRANSFER;
  Transfer(n,M,Boat,b) is Disembark(n,b);

import MOVE;
  Move(Boat,p) is CrossTo(p);

import NOCONCURRENCY;

axioms
  Location(Bank1)=P1;
  Location(Bank2)=P2;

  % The boat can carry at most five
  constraint -(Amount(M,Boat)=n & MaxBoatCapacity < n);
```

```
nonexecutable CrossTo(p) if Amount(M,Boat)=0;
```

Using numeric resources instead of individuals allows us to easily state the axiom about having no more missionaries than the boat capacity. (Recall that when we formalized the domain using individual missionaries, we relied on the boat capacity (2) and the total number of missionaries (3) being separated by one. Therefore the axiom in that version stated that "not all missionaries are allowed on the boat together," rather than a condition about a specific boat capacity.)

Sample CCalc Query for Missionaries as Resources

We asked CCalc to solve the missionaries problem with 9 missionaries starting out on `Bank1` initially and ending up on `Bank2` eventually. (When processing the MAD description above, the library symbol `MaxAmount` was set to `9`.)

```
:- query
maxstep :: 6..9;
0: amount(m,bank1)=9,
   amount(m,bank2)=0,
   amount(m,boat)=0;
maxstep:
   amount(m,bank1)=0,
   amount(m,bank2)=9,
   amount(m,boat)=0.
```

The shortest plan takes 7 actions:

```
0:  board(5, bank1)
1:  crossto(p2)
2:  disembark(4, bank2)
3:  crossto(p1)
```

```
4:  board(4, bank1)
5:  crossto(p2)
6:  disembark(5, bank2)
```

11.5 Missionaries and Cannibals

In Section 9.4 we considered a simplified version of the original missionaries and cannibals domain, with only missionaries, because formalizing both missionaries and cannibals as individuals becomes quite cumbersome. Now that we can reason with groups, we present a formalization of the original missionaries and cannibals puzzle, with 3 missionaries and 3 cannibals.

This formalization is similar to the formalization with only missionaries, shown in the preceding section, but it differs in a few ways, due to having cannibals. Actions `Board` and `Disembark` have one more argument, because there are now two kinds of resources, missionaries and cannibals. The axiom about the boat capacity changes because there may now be passengers of two kinds. We also add the condition from the original puzzle that the missionaries must never be outnumbered by the cannibals — or else they'll be eaten.

```
include "../library-ontology"
include "../library"

sorts
   RiverBank;

inclusions
   RiverBank << Accumulator;

% Maximum number of missionaries/cannibals
numeric_symbol MaxMiss=3
```

```
% Maximum capacity of the boat
numeric_symbol MaxBoatCapacity=2

module MC;

  objects
    M, C : Resource;
    P1, P2 : Place;
    Bank1, Bank2 : RiverBank;
    Boat : Accumulator;

  actions
    Board(Resource, 1..MaxMiss, RiverBank);
    Disembark(Resource, 1..MaxMiss, RiverBank);
    CrossTo(Place);

  variables
    n     : 1..MaxMiss;
    n1,n2 : 0..MaxMiss;
    r     : Resource;
    b     : RiverBank;
    p     : Place;

  import TRANSFER;
    Transfer(n,r,b,Boat) is Board(r,n,b);

  import TRANSFER;
    Transfer(n,r,Boat,b) is Disembark(r,n,b);

  import MOVE;
    Move(Boat,p) is CrossTo(p);

  import NOCONCURRENCY;
```

```
  axioms
    Location(Bank1)=P1;
    Location(Bank2)=P2;

    % The boat cannot carry more than its capacity
    constraint -(Amount(M,Boat)=n1 & Amount(C,Boat)=n2
                 & MaxBoatCapacity < n1+n2);

    nonexecutable CrossTo(p) if Amount(M,Boat)=0 & Amount(C,Boat)=0;

    % Cannibals should never outnumber missionaries
    constraint -(Amount(M,b)!=0 & Amount(M,b) < Amount(C,b));
    constraint -(Amount(M,Boat)!=0 & Amount(M,Boat) < Amount(C,Boat));
```

Sample CCalc Query for Missionaries and Cannibals

We asked CCalc to solve the missionaries and cannibals problem with 3 missionaries and 3 cannibals starting out on `Bank1` initially and ending up on `Bank2` eventually. (When processing the MAD description above, the library symbol `MaxAmount` was set to 3.)

```
:- query
maxstep :: 18..19;
0: amount(m,bank1)=3,
   amount(c,bank1)=3,
   amount(m,bank2)=0,
   amount(c,bank2)=0,
   amount(m,boat)=0,
   amount(c,boat)=0;
maxstep:
   amount(m,bank1)=0,
   amount(c,bank1)=0,
```

```
   amount(m,bank2)=3,
   amount(c,bank2)=3,
   amount(m,boat)=0,
   amount(c,boat)=0.
```

The shortest plan takes 19 actions:

```
0:  board(c, 2, bank1)
1:  crossto(p2)
2:  disembark(c, 1, bank2)
3:  crossto(p1)
4:  board(m, 1, bank1)
5:  crossto(p2)
6:  disembark(m, 1, bank2)
7:  crossto(p1)
8:  board(m, 1, bank1)
9:  crossto(p2)
10:  disembark(m, 1, bank2)
11:  crossto(p1)
12:  board(m, 1, bank1)
13:  crossto(p2)
14:  disembark(m, 1, bank2)
15:  crossto(p1)
16:  board(c, 1, bank1)
17:  crossto(p2)
18:  disembark(c, 2, bank2)
```

11.6 A New Library Module: BUY

Buying and selling objects [Lee and Lifschitz, 2006] can also be modeled using resources and the library module TRANSFER. The module below specializes the notion of transfer to cases where there is a buyer and a seller and commodities are transferred in exchange for money.

For this, we add the following declarations to our library ontology:

```
sorts
   Commodity;
   Buyer;
   Seller;

inclusions
   Commodity << Resource;
   Buyer << Accumulator;
   Seller << Accumulator;
```

As a result of adding these declarations, the sort and subsort relations in the library ontology now become as shown in Figure 11.2.

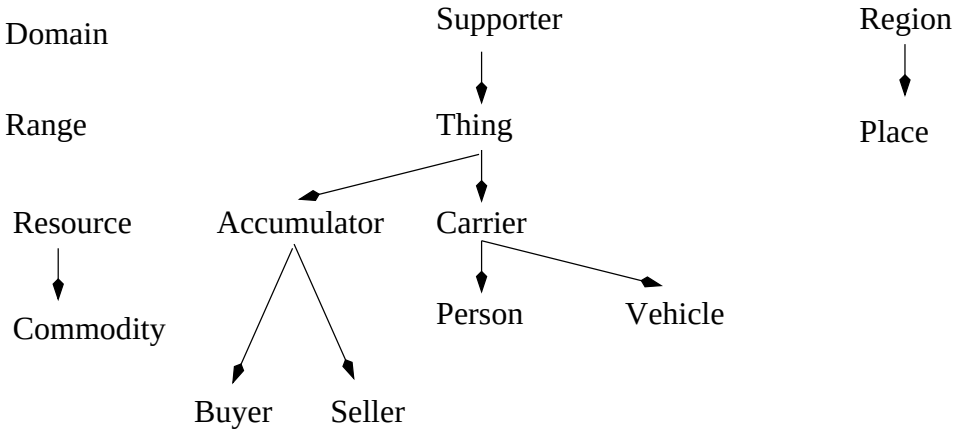

Figure 11.2: Sorts and subsort relations in ontology after adding module BUY

```
module BUY;

  objects
    Money   : Resource;

  actions
    Buy(Buyer, 1..MaxAmount, Commodity, Seller, 1..MaxAmount);
```

```
fluents
  Price(Commodity) : rigid(1..MaxPrice);

variables
  n    : 1..MaxAmount;
  n2   : 1..MaxPrice;
  c    : Commodity;
  b    : Buyer;
  s    : Seller;
  x    : Thing;
  cost : 1..MaxAmount;

import TRANSFER;
  Transfer(n,c,s,b) is Buy(b,n,c,s,cost);

import TRANSFER;
  Transfer(cost,Money,b,s) is Buy(b,n,c,s,cost);

axioms
  nonexecutable Buy(b,n,c,s,cost) if Price(c)=n2 & cost!=n*n2;
```

One limitation of this formalization comes from the fact that the version of MAD we used in this dissertation restricts the right-hand side of a constant renaming `is` statement to be a single constant, not a complex formula. If we didn't have this restriction, we could perform "conditional renamings" of actions: Instead of having to include the cost of a buying action as an argument, we could reduce the arguments to action `Buy` and have the calculation of the cost be part of the second renaming statement by writing

```
import TRANSFER;
  Transfer(cost,Money,b,s) is Buy(b,n,c,s) & cost=n*Price(c);
```

If we were able to write such a renaming statement, the only axiom in the module could be dropped too.

Another limitation of module BUY is one that it inherits from module TRANSFER: concurrent transfers are not allowed. As we mentioned in Section 11.3, the solution to this problem is to enhance MAD with "additive fluents."

Both of these limitations are listed among topics for future work in Section 15.2.1.

11.7 Buying Flowers

Here we formalize a specific example of a buyer, John, buying flowers from a store.

```
include "../library-ontology"
include "../library"

module BUYING_FLOWERS;

  import BUY;

  objects
    Flowers : Commodity;
    Austin  : Place;
    Store1  : Seller;
    John    : Buyer;

  variables
    x          : Thing;

  import NOCONCURRENCY;

  axioms
    Location(Store1)=Austin;
```

```
    Location(John)=Austin;
    inertial Location(x);

    Price(Flowers)=2;
```

Sample CCalc Query about Buying Things

Instead of a planning problem, we asked CCalc to solve a projection query: John has $10 and there are 10 flowers in the store. If he buys 3 at $2 each, how many flowers and how much money will he have? (When processing the MAD description above, the library symbol `MaxAmount` was set to `10` and `MaxPrice` was set to `5`.)

```
:- query
maxstep :: 1;
label :: 0;
0: amount(money,john)=10,
   amount(money,store1)=3,
   amount(flowers,john)=0,
   amount(flowers,store1)=10,
   buy(john,3,flowers,store1,6).
```

We only specified the initial state and one action above, instructing CCalc to find a projection for the next state. It told us that John has 3 flowers and $4 left. Also, the store's money increased by $6 and its inventory of flowers decreased by 3.

```
0:  amount(money, john)=10  amount(money, store1)=3
    amount(flowers, john)=0  amount(flowers, store1)=10
    location(john)=austin  location(store1)=austin

    ACTIONS:  buy(john, 3, flowers, store1, 6)

1:  amount(money, john)=4  amount(money, store1)=9
    amount(flowers, john)=3  amount(flowers, store1)=7
```

location(john)=austin location(store1)=austin

Chapter 12

Further Enhancements to the MAD Language

In Chapter 6 we described new features we added to the MAD language. In addition to those new features, we also restricted the input language of our implementation in some ways and modified the original semantics of MAD. Here we describe these enhancements to MAD.

12.1 Changes for Transformation into other Implemented Languages

In order to implement the MAD language as quickly as possible, we decided to leverage the power of CCalc. However, CCalc captures only the "definite" fragment of action language $\mathcal{C}+$ but constant renaming statements in MAD correspond to nondefinite causal laws. In some cases, it is possible to use a simple transformation to turn an action description containing such nondefinite causal laws into one

which contains only definite laws, though it is not always straightforward to identify whether a given nondefinite description falls under these cases. Experimenting with manual transformation of automatically-generated nondefinite CCalc input into definite form, we noticed that all of our library modules and specific domain formalizations could be expressed in a form in which this tranformation would be applicable. Therefore we restricted the input language we use to guarantee that all nondefinite causal laws will fall under this case, which allowed us to automate the transformation into a definite action description. We present the restrictions below:

- *A single atom instead of a formula in renamings*: Here is an example of a MAD `import` section, which includes a constant renaming statement, redefining constant `Move` in terms of constant `Carry`.

  ```
  import MOVE;
    Move(x, p) is Carry(u, x, p);
  ```

  The intuitive meaning of this is "the action of agent `u` carrying object `x` to place `p` is synonymous with the action of moving `x` to `p`." We refer to the formulas on either side of the keyword `is` as the left-hand side and the right-hand side of the renaming statement.

  In the general description of MAD, the left-hand side of a constant renaming statement consists of the constant being renamed but the right-hand side allows a full formula. There are two problems with this: (i) it is difficult to check whether the resulting axioms constitute an "explicit definition" of the renamed constant (as required by MAD), and (ii), the "nondefinite" axioms corresponding to it are too complex to automatically convert into "definite" causal laws, as is required by the Causal Calculator. On the other hand, we

observed that, in practice, all of the renaming statements we encountered in early versions of the library modules and example domains were writable in a form in which at most a single atom appeared on the right-hand side. In such cases, it is possible to apply a simple procedure suggested in Chapter 4 to convert these nondefinite action descriptions into definite ones. Therefore the right-hand side of a constant renaming statement is now restricted to have a single atom or the zero-place connective `false`. The axioms resulting from such renaming statements are called "semi-definite." Using this form in MAD guarantees that the axioms generated from renaming statements are explicit definitions and it enables us to automatically transform the resulting action description into the language of CCalc.

- *Cases in renaming statements*: In the original definition of MAD renaming statements, all of the arguments to the constant on the left-hand side had to be variables of the argument sort from the declaration of the constant. If we wanted to redefine instances of that constant with different parameters differently, we would need a complex formula on the right-hand side, even if this formula were reducible to a single atom for each different value tuple of the arguments. Here is an example of such a complex formula:

```
import CLIMB;
  Climb(u,s) is u=Monkey & ((s=Box & ClimbOn)
                          | (s=Floor & ClimbOff));
```

The intuitive meaning of the constant renaming statement above is "the action of `Climb` is always executed by the monkey and is synonymous with action `ClimbOn` if climbing onto the box, or with action `ClimbOff` if climbing onto the floor."

Since the restriction described in the preceding bullet allows only single contants on the right-hand side, we needed a way to conveniently differentiate between different "cases" of argument values. Therefore we allow the right-hand side to be broken up into several "case statements," with each case having a condition formula without any constants, and a single atom (or `false`) redefining the renamed constant. The example can be written as

```
import CLIMB;
  Climb(x,s) is
    case x=Monkey & s=Box : ClimbOn;
    case x=Monkey & s=Floor : ClimbOff;
```

Note that both cases say that "`x=Monkey`". To make expressing this a little more convenient we allow objects, and also variables of subsorts of the sort expected by the constant, as arguments to constants on the left-hand side of a renaming statement. The example above is now written as

```
import CLIMB;
  Climb(Monkey,s) is
    case s=Box : ClimbOn;
    case s=Floor : ClimbOff;
```

We will say more about the precise semantics of case statements in the next section.

12.2 Modifications to the Semantics of MAD

In Chapter 5 we described how imported modules are incorporated into later ones that refer to them. During this process, whenever a constant is redefined, the

imported copy is renamed by prepending a prefix of the form "In." to it.[1] The result of applying this process repeatedly is a single-module action description, which corresponds to an action description in $\mathcal{C}+$.

As we mentioned in Chapter 6, one of the enhancements we made to the original syntax of MAD, the ability to have more variables on the right-hand side of a constant renaming clause, requires a modification of the import semantics. In addition, as we worked on writing modules in the MAD language, we identified some more problematic issues in the proposed semantics of import statements. These issues become important when there are multiple levels of imports. (Which may explain why they weren't noticed during the initial design of MAD.)

Here are the changes to the semantics that were made as a result of our early attempts at writing modules:

- *When a constant renaming statement has variables on its right-hand-side that don't appear on its left-hand side, the renamed version of the constant on the left-hand side must be given extra arguments.*

  Consider the following example of this kind of renaming statement,

```
sorts
  Agent;
  Thing;
  Place;

module MOVE;
  actions
    Move(Thing,Place);
```

[1]Here I stands for import and n will be an integer depending on the number of imports that have been processed so far.

```
module CARRY;

  actions
    Carry(Agent,Thing,Place);

  variables
    u : Agent;
    x : Thing;
    p : Place;

  import MOVE;
    Move(x, p) is Carry(u, x, p);
```

The intuitive meaning of this is "the action of agent `u` carrying object `x` to place `p` is synonymous with the action of moving `x` to `p`." Without any change to the semantics, this import statement adds the axiom

```
I1.Move(x, p) <-> Carry(u, x, p);
```

Now, imagine having two agents, `Alice` and `Bob`. Then, the equivalence above would entail

```
I1.Move(x, p) <-> Carry(Alice, x, p);
I1.Move(x, p) <-> Carry(Bob, x, p);
```

and hence

```
Carry(Alice, x, p) <-> Carry(Bob, x, p);
```

This is clearly not what we want. In order to distinguish between actions executed by different agents `u`, the renamed constant `I1.Move` should get a third argument, of the sort of variable `u` above. The action declaration of

`I1.Move` becomes

```
action
  I1.Move(Thing,Place,Agent);
```

and the equivalence axiom becomes

```
I1.Move(x, p, u) <-> Carry(u, x, p);
```

Now, `Alice` and `Bob` can carry objects independently.

In addition to the declaration of the constant changing, all of occurrences of this constant need to be changed, adding new variables to occurrences in the axioms. This involves declaring new variables, in order to ensure the new arguments in the constant occurrences are not the same as other variables in the formula.

- *If a constant c being renamed is given extra arguments (as in the bullet above), then these new arguments should be propagated down to all constants renamed in terms of c, and this propagation should be repeated recursively for all those constants too.*

  The change to semantics from the preceding bullet solved the issue of `Alice` and `Bob` carrying things independently. However, the same problem can still occur if we rename action `Move` in two steps. (Using two levels of imports.) Consider modifying the example above by adding another module in between `MOVE` and `CARRY`:

```
sorts
  Agent;
  Thing;
```

```
  Place;

module MOVE;

  actions
    Move(Thing,Place);

module TAKE;

  actions
    Take(Thing,Place);

  variables
    u : Agent;
    x : Thing;

  import TAKE;
    Move(x, p) is Take(x, p);

module CARRY;

  actions
    Carry(Agent,Thing,Place);

  variables
    u : Agent;
    x : Thing;
    p : Place;

  import TAKE;
    Take(x, p) is Carry(u, x, p);
```

The first import introduces the axiom

```
I1.Move(x, p) <-> Take(x, p);
```

The second import changes the declaration of `Take` and introduces the axiom

```
I2.Take(x, p, u) <-> Carry(u, x, p);
```

In addition, the occurrence of `Take` in first axiom is modified:

```
I1.Move(x, p) <-> I2.Take(x, p, u);          (12.1)
```

But this is not enough to ensure independence of `Alice` and `Bob` carrying things, because, similarly to the example above, but in two steps, we get

```
Carry(Alice, x, p) <-> Carry(Bob, x, p);
```

The solution to this problem is to propagate any additional arguments to all constants which have been renamed in terms of constants receiving new arguments. This way, `I1.Move` would get an additional agent argument as an effect of the constant renaming in the second import, and axiom (12.1) becomes:

```
I1.Move(x, p, u) <-> I2.Take(x, p, u);
```

- *When constants in an imported module are being renamed, a prefix "*`In.`*" should be prepended not only to constants being renamed in the current import, but also to constants with at least one prefix "*`Im.`*"*

Another problem arises with multiple levels of imports, when the same module is imported more than once with different renamings. Consider modifying the example above, where there are two levels of imports, by adding a second import statement in the last module:

```
sorts
  Agent;
  Thing;
  Place;

module MOVE;

  actions
    Move(Thing,Place);

module TAKE;

  actions
    Take(Thing,Place);

  variables
    u : Agent;
    x : Thing;

  import TAKE;
    Move(x, p) is Take(x, p);

module CARRY;

  actions
    Carry(Agent,Thing,Place);
    Transfer(Agent,Thing,Place);
```

```
variables
  u : Agent;
  x : Thing;
  p : Place;

import TAKE;
  Take(x, p) is Carry(u, x, p);

import TAKE;
  Take(x, p) is Transfer(u, x, p);
```

In accordance with the new semantics in the preceding bullets, the import of `MOVE` into `TAKE` introduces the axiom

```
I1.Move(x, p) <-> Take(x, p);
```
(12.2)

into module `TAKE`.

The first import of `TAKE` introduces the axiom

```
I2.Take(x, p, u) <-> Carry(u, x, p);
```
(12.3)

and also brings in the modified version of (12.2):

```
I1.Move(x, p, u) <-> I2.Take(x, p, u);
```
(12.4)

The second import of `TAKE` introduces the axiom

```
I3.Take(x, p, u) <-> Transfer(u, x, p);
```
(12.5)

and also brings in the modified version of (12.2):

```
I1.Move(x, p, u) <-> I3.Take(x, p, u);          (12.6)
```

Now, since the renaming works by prefixing "In." to the constant renamed in the immediate import, the two imported versions of action `Take` have different prefixes. However, the action `I1.Move`, in terms of which these actions are defined, (and whose name resulted from the renaming in the first import in the action description) has not been renamed during the two imports of `TAKE`. Thus axioms (12.4) and (12.6) entail

```
I2.Take(x, p, u) <-> I3.Take(x, p, u);
```

and, along with (12.3) and (12.5), entail

```
Carry(x, p, u) <-> Transfer(x, p, u);
```

The solution to this problem is to rename any constants whose names include at least one prefix "In." (The string of prefixes in every constant name will provide a complete description of the path that the name followed in the sequence of imports.) With this change, axiom (12.4) becomes

```
I2.I1.Move(x, p, u) <-> I2.Take(x, p, u);
```

and axiom (12.6) becomes

```
I3.I1.Move(x, p, u) <-> I3.Take(x, p, u);
```

The actions `Carry` and `Transfer` are now independent, as should be.

- *Case statements in renamings*:

  One of the enhancements we made to the original syntax of MAD is for import sections with case statements. A renaming clause with case statements is of the form

  $c(t_1, ..., t_n)$ `is`
  `case` F_1 `:` F'_1 `;`
  $\vdots$
  `case` F_k `:` F'_k `;`
  `default` `:` F'_{k+1} `;`

  where c is the constant being renamed, $t_1, ..., t_n$ are variables of the module where the import occurs, and $n, k \geq 0$. Each variable must be of the sort declared for the corresponding argument of c and all variables must occur at most once. Each F_i must have no constants and should not contain any variables other than those in $t_1, ..., t_n$. Each F'_i must be a single constant with the same domain as c, or an object from the domain of c. The last case, with "`default`" stands for the case where F_{k+1} is the zero-place connective $\top$. If c is Boolean-valued, this last case may be omitted and it is treated as if F'_{k+1} were $\bot$.

  The equivalence introduced by such a renaming statement is (assuming this is the mth import in the action description)

$$\mathrm{I}m.c(t_1, ..., t_n) \equiv \bigwedge_{i=1}^{k} (\bigwedge_{j=1}^{i-1} \neg F_j \wedge F_i \supset F'_i) \tag{12.7}$$

For each value of $t_1, ..., t_n$, the formula on the right-hand side will be logically equivalent to exactly one F'_i. This equivalence guarantees an "explicit definition" of each instance of $\mathtt{I}m.c$.

Recall that, the enhancements to the syntax relax the requirement of having $t_1, ..., t_n$ be variables of sorts matching the declaration of constant c. Instead, we may have objects of the declared sort, or variables which belong to a subsort of the sort required by the declaration. In such cases, all instances of the constant which do not match this restricted form should be equivalent to $\bot$. In other words, the above equivalence (12.7) is modified to be

$$\mathrm{I}m.c(v_1, ..., v_n) \equiv ((G_1 \vee ... \vee G_n) \supset \bot) \wedge \bigwedge_{i=1}^{k} (\neg(G_1 \vee ... \vee G_n) \wedge \bigwedge_{j=1}^{i-1} \neg F_j \wedge F_i \supset F'_i) \tag{12.8}$$

where $v_1, ..., v_n$ are variables of the sort required by the declaration of c, and G_i stands for

- $v_i \neq t_i$ if t_i is an object,
- $\neg Sort_{t_i}(v_i)$, if t_i is a variable, where $Sort_{t_i}$ is the "sort name predicate" of the sort of variable t_i.

This covers all instances of c which are not covered by $t_1, ..., t_n$.

Chapter 13

Using the MAD Implementation

As part of this dissertation, we developed an implementation of the MAD language, with the extensions described in Chapters 6 and 12. This implementation was used constantly to test all our formalizations in MAD. Such tests serve to increase our confidence both in the adequacy of the formalizations and in the soundness of the implementation of MAD.

In this chapter we explain how one may obtain, set up, and use the MAD implementation. A detailed description of the allowed input may be found in Appendix D.

A note about fonts

In the scope of this chapter, things written in `typewriter font` indicate input that should appear exactly as shown. Things written in *italics* refer to an element of input which has already been explained or is about to explained.

13.1 Obtaining MAD and System Requirements

MAD can be downloaded from its homepage:

`http://www.cs.utexas.edu/~tag/mad/`

The latest version available is MAD version 0.4.

The software is written in the C programming language. In order to build MAD from source, one needs a C compiler, a Lex-like lexical analyzer generator, and a Yacc-like parser generator.

We have tested our software on Unix/Linux-like systems and used the following programs to build MAD from source:

- GCC, the GNU Compiler Collection
- flex, The Fast Lexical Analyzer
- bison, the GNU parser generator

More specifically, we have successfully tested MAD with the following combinations:

- Debian GNU/Linux (sid), gcc 4.2.3, flex 2.5.35, bison 2.3
- SunOS 5.9, gcc 4.2.2, flex 2.5.33, bison 2.0

13.2 Building and Running MAD

13.2.1 Building the Code

As mentioned in the first section, we used the tools gcc, flex and bison on Unix/Linux systems.[1] Once these are installed, to build the program from source, simply run the command `make` at the shell prompt. This will produce an executable named `mparse`.

13.2.2 Running the Program

`mparse` is a program which parses its MAD input and processes the import statements, building internal data structures corresponding to each module in the input, finally printing these processed modules to an output file.

The programs expects to be called as follows:

$$\texttt{mparse}\ [-\texttt{i}]\ \textit{input\_file}_1\ \textit{input\_file}_2\ \ldots\ \textit{input\_file}_n\ [-\texttt{o}\ \textit{output\_file}]$$

where the arguments within square brackets are optional. If no output file is specified, the output is written to a file named `mparse.output`.

When more than one input file is specified, `mparse` treats them as if they were all concatenated in one big file. The optional switches `-i` and `-o` may occur in any place in the command line, not just at the beginning or the end. However, the `-o` switch must be followed by an output file.

[1] If the user wishes to replace gcc, flex, or bison with other software that accomplishes the same job, he will need to change the makefile.

13.2.3 MAD Output: CCalc Input

By default, `mparse` automatically generates a CCalc input file corresponding to the final module in the MAD action description.

It is important to note the following: CCalc doesn't allow identifiers to have free capitalization, so the output of `mparse` will turn all letters in identifiers into their lower-case form. The program will print a warning message if two identifiers become the same when converted to lower-case.

13.2.4 Other Forms of Output

If the program is called with the `-i` (interactive) option, then after parsing, instead of directly generating CCalc code, the user is asked to choose among different options to view the resulting modules:

1. Print modules parsed: prints the data structures built for all of the modules, with the import statements processed.

2. Print last module: prints only the data structures built for the last module in the MAD description, with the axioms grounded.

3. Print last module as CCalc input: like the preceding option, but prints the module in CCalc format.

4. Quit without printing anything.

13.3 Using MAD with CCalc

13.3.1 Running CCalc on MAD Action Descriptions

We don't explain the details of running CCalc here. For that, the reader is referred to the CCalc homepage:

`http://www.cs.utexas.edu/~tag/cc/`

As explained in the preceding section, the MAD executable `mparse` can turn a given MAD action description into a CCalc input file, albeit with the identifiers turned into all-lower-case words. Therefore, when writing a CCalc query to be used with the given file, one must write all identifiers as all-lower-case.

In the course of developing an action description, it is often the case that a user first runs `mparse` on a MAD description, prepares a set of CCalc queries, loads both of them into CCalc and tests them, only to find that he has to go back to make some changes to the MAD description, and tests with the same queries. In order not to change any files other than the MAD input file, we recommend using the CCalc include statement

`:- include '`*ccalc-input-filename*`'.`

at the beginning of files containing queries. This is the approach we have taken in the examples provided as part of the MAD distribution. Each CCalc query file, (e.g. `bw-queries`) begins with a CCalc include statement specifying the name of the file we expect `mparse` will generate (e.g. `bw.cc`). Then, when running `mparse` we use the `-o` option to tell it to generate an output file with this name (e.g. `mparse`

`bw -o bw.cc`).

13.3.2 Issues to Watch Out for when Running MAD with CCalc

- The CCalc file generated by `mparse` includes a `show` specification which hides certain constants from the output. Those hidden are the renamed constants beginning with an import prefix of the sort "`I`n`.`" and also the constants "`Actor`" and "`Theme`" which are declared in the MAD library. (These will be hidden even if the user declares them himself, without reference to the library.)

- One must make sure that there are no similar identifiers with different capitalization. In MAD, `P2` and `p2` are different identifiers because MAD is case-sensitive. But during the automatic conversion of MAD files into CCalc input, they'll both turn into `p2`. The MAD executable `mparse` will give a warning in such cases but will still print out a CCalc input file. (If one uses such a problematic file with CCalc, such errors are very hard to figure out because CCalc won't complain even if the same identifier is declared as both an object and a variable.)

13.4 Debugging Action Descriptions

Common Errors in Formalizing Action Domains

Often the first attempt to formalize an action domain is not successful and running CCalc yields no solution. In such cases the first thing to do is to run two simple queries to check whether the transition system corresponding to the description has any valid states and transitions.

```
% Tests whether the transition system has any valid states
```

```
:- query
maxstep :: 0;
label :: 0.

% Tests whether the transition system has any valid transitions
:- query
maxstep :: 1;
label :: 1.
```

If the first query succeeds but the second fails, this means that even though there are causally explained states in the transition system, no causally explained transition exists. This is most often because an action instance has no cause specifying whether it should be executed or not. Usually we make action constants exogenous and don't specify any further cause for them. So a common mistake is to forget to make an action constant exogenous.

A related mistake occurs when we import another module and define a new action in terms of one from the imported module. If the old action was declared to be exogenous we usually won't specify that the new one is exogenous too, because that follows from the two actions being equivalent. However, if the equivalence does not cover all instances of the new action, then a cause must be explicitly specified for these other instances. For example, the following action description snippet declares a new sort `Container` which is a subsort of the library sort `Thing`, and a new action `PutIn` is defined in terms of library action `Mount`.

```
  inclusions
    Agent << Thing;
    Container << Thing;
```

```
module CONTAINER;
  constants
    PutIn(Thing, Container) : action;

  variables
    x : Thing;
    c : Container;

  import MOUNT;
    Mount(x,c) is
      case -Agent(x) : PutIn(x,c);
```

Here the import statement specifies that the two actions are equivalent only when their first arguments are not of library sort `Agent`. So, even though action `Mount` is exogenous, instances of `PutIn` where the first argument is an `Agent` don't inherit this property.

13.4.1 Typical Mistakes when Using Library Modules

- Using module `LOCAL` for actions which don't have any `Actor` or `Theme` defined.
- Forgetting to include module `TOP` for domains which have both a concept of support and a concept of locations involved, or forgetting to define a `TopLocation` value for all `Things`.

Chapter 14

Implementation Details

Our implementation of MAD that allows us to perform various kinds of reasoning (such as planning, prediction, postdiction) about action descriptions written in MAD. The implementation makes use of the Causal Calculator (CCalc) which is a system that can reason with the "definite" fragment of language $\mathcal{C}+$.

The system takes as input the library of basic action descriptions along with a domain-specific action description. It first turns this set of modules into an equivalent single-module description by eliminating import statements and incorporating their contents, with appropriate modifications, into the importing module, according to the semantics given in [Lifschitz and Ren, 2006] and modified as shown in Chapter 12. A module without import sections is essentially a $\mathcal{C}+$ description. However, this description generally contains "nondefinite" axioms that CCalc cannot handle. Therefore we need to apply a further transformation, based on the methods outlined in Chapter 4, which turns the description into an equivalent definite description. The final output is an action description which can be fed into CCalc.

In this chapter we briefly describe various aspects of the implementation.

14.1 Implementing Import Unrolling

The fundamental feature of MAD is the ability to import earlier modules into newer ones. In the original semantics of MAD, the process of incorporating a module imported into another consists of merging declarations and axiom sections of the two modules. Constant renaming statements turn into equivalence axioms, and an appropriate prefix "Im." is prepended to all occurrences of the redefined constant. No other record of renaming statements is kept, other than the equivalence axiom.

After making the semantics modifications dealing with multiple levels of imports (described in Chapter 12), it became necessary for the importing procedure α to add a "hidden" section to modules in order to properly propagate the renamings down to the previously-imported modules. This new section contains an ordered (newer-to-older) list of renamings carried out to obtain the current form of the module. Such "hidden" sections provide necessary information during a sequence of imports, though they do not have any effect as long the module in which they appear is not imported further. Therefore, they may be disregarded after turning an action description into a single module.

14.2 Making Nondefinite Action Description Definite

As mentioned earlier in this chapter, the semantics of MAD describes how to turn a MAD action description into a family of action descriptions in $\mathcal{C}+$. Even with the restrictions to the input language listed in Chapter 12 (such as import statements allowing only a single constant on the right-hand side, instead of a full formula), these action descriptions may still be "semi-definite," and thus not acceptable by CCalc, which requires a definite action description. Fortunately, for semi-definite

action descriptions, where the semi-definite axioms arise purely from explicit definitions, there exists a procedure which can convert this semi-definite description to a definite one.

14.2.1 Replacing Renamed Constants

Consider the renaming statement

```
import MOVE;
  Move(x, p) is Carry(u, x, p);
```

This would turn into the following $\mathcal{C}+$ axiom (expressed in the language of MAD):

```
In.Move(x, p, u) <-> Carry(u, x, p);
```

This axiom constitutes an explicit definition of `In.Move(x, p, u)` in terms of `Carry(u, x, p)`. It is semi-definite because an equivalence between two atoms appears in the "head" part of the axiom. By Proposition 1 in Chapter 4, in the presence of such an axiom expressing equivalence, we may replace any other occurrence of `In.Move(x, p, u)` by `Carry(u, x, p)` without changing the meaning of the action description. On the other hand, by Proposition 3 of Chapter 4 if the action description does not contain any other occurrences of constant `In.Move(x, p, u)`, then we can replace its explicit definition by the following definite axiom:

```
In.Move(x, p, u) if Carry(u, x, p);
```

This axiom is definite because there is only a single constant in the "head" (the part before the `if`).

The procedure below takes a semi-definite action description obtained from a MAD action description, and converts it into a definite action description. The

semi-definite action description obtained from processing a MAD action description is assumed to contain an ordered (newer-to-older) list of renamings carried out to obtain the current form of the module. (Our implementation produces this list during the process of incorporating `import` sections, and adds it to each processed module as a "hidden" section. These "hidden" sections provide information which is necessary to properly propagate the renamings down to the previously-imported modules, as required by the new semantics.)

Algorithm 1 Takes a semi-definite action description obtained from a MAD action description and converts it into a definite action description

1: Reverse the renaming list so that the oldest renaming appears first.
2: **for all** renamings in the renaming list **do**
3: Use equivalence axiom corresponding to renaming, to replace occurrences of the renamed constant in heads of non-equivalence axioms
4: Replace the equivalence axiom for the renamed constant by a definite law
5: **end for**

The soundness (in terms of preserving the meaning of the action description) of the algorithm above is guaranteed by Propositions 1 and 3 of Chapter 4. The statement on line 3 depends on Proposition 1 and the statement on line 4 depends on Proposition 3.

14.2.2 Multi-Sorted Unification

In order to carry out the algorithm above, where occurrences of a renamed constant in the heads of laws are replaced by another constant from the equivalence axiom, we need to determine if the constant in the head matches the constant in the equivalence axiom. And since both the constant in the head and the constant in the equivalence axiom are written in a schematic form (using variables) we need to perform unification to determine which instances to replace.

The basic unification algorithm [Robinson, 1965] works on non-sorted variables/objects. However, MAD is a sorted language so our unification algorithm accounts for sorted variables/objects, along with integers and integer ranges.

Similar to standard unification, we begin by "standardizing apart" the variables in the two terms we wish to unify. After that, this is how we unify arguments of constant terms:

- if both are objects, they must be the same.
- if one argument is a variable and the other an object, then the object must be of the sort of the variable.
- if the two arguments are both variables, then they must be of the same sort, or must have a common subsort. In the latter case, a new variable of the common subsort will be used for replacement.
- if both arguments are integers, they must be the same.
- if one argument is an integer and the other is an integer range, then the integer must lie within the range.
- if the two arguments are both integer ranges, then they must overlap. The intersection range is will be used for replacement.

14.3 Grounding

After making the action description definite, one more significant step is needed before a file in the language of CCalc can be produced. MAD has the built-in sort `explicitAction`, which doesn't exist in CCalc. This poses a problem for grounding (the process of eliminating variables, whereby all axioms with variables

are replaced by possibly many copies, each with a different assignment of values to the variables), since we want variables of the sort `explicitAction` to be replaced by only explicitly declared actions. In order to bypass this problem, our implementation grounds the action description itself. (This grounding is very fast compared to the grounding in CCALC and thus does not bring any performance penalty.)

14.3.1 Grounding Argumented Objects

In order to do grounding of variables in the axioms, we need to have a list of all objects belonging to each sort. One challenging aspect of grounding is how to handle objects with arguments. Without arguments, at the end of the action description we would have a list of objects and we cound simply go down the list of objects and assign each one to its sort (and supersorts). However, when an object declaration has an argument, in order to know what objects that corresponds to, we need to know all objects that belong to the sort of any arguments. For example, with the declarations

```
sorts
  Building; Person;

objects
  John, Bob     : Person;
  House(Person) : Building;
```

we need to know what the objects of sort `Person` are before we can know the objects of sort `Building`.

The approach we take is to make a forest of "sort dependencies." Sort dependencies work as follows:

- if we have a pair of object declarations

```
objects
  a(s1, s2) : s
  b(s4, s5) : s3
```

then this will add edges "s → s1," "s → s2," "s3 → s4," "s3 → s5," to the sort dependency forest.

- if we also have an inclusion "s3 << s" then this means that all objects of s3 are also objects of s, so everything that s3 depends on must also be depended on by s. The forest will get the additional edges "s → s4", "s → s5".

In order to make a list of objects of each sort, we have a procedure that takes a sort and discovers all of the objects of that sort. If the sort has "sort dependencies", then the procedure recursively calls itself on all of the dependencies. This ensures that he object lists are built in the right order. (Built-in sorts Boolean and integer ranges have their objects built before any other declared sorts.)

No cycles are allowed in the sort dependency forest because if there is a cycle in the sort dependency graph, then that would mean that a sort can have an infinite number of objects. (Because whenever there's a new object of sort s, that will lead to a new object of some sort that is depended upon by s, and then there will be yet another object of sort s due to the cycle. This process will repeat forever.)

The implementation keeps track of the sort dependency forest as it reads in the input and reports an error when it detects a cycle.

14.3.2 Grounding Integers

Since integer ranges are built-in sorts and integers are built-in objects, they must be taken into account whenever we make a list of objects of all sorts. Every integer

range that occurs in an action description is turned into a built-in sort and all of the integers that fall within that range are listed as objects of that sort.

14.4 Automatic Translation into the Language of CCalc

Once grounding is done, the final step of producing an input file for CCalc consists of making a few simple syntactic changes to conform to the syntax of CCalc.

The process of conversion to CCalc format turns all MAD identifiers into their lower-case equivalents. Therefore two identifiers which differ only in capitalization turn into the same identifier in CCalc format. The system prints out a warning message in such cases.

Changes to CCalc for Better Output

In the process of implementation we made minor changes to CCalc code too.

- CCalc normally shows values of all constants at each time step. Since we are usually only interested in explicitly declared (i.e. not renamed) constants, we generate a CCalc `show` command that suppresses all constants that begin with `I`n..
- The CCalc `show` command didn't originally affect action constants so we modified CCalc so that our suppression of implicitly declared constants covers action constants too.
- In addition to suppressing implicitly declared constants, we also suppress some library fluents which take actions as arguments (and hence have unwieldly looking implicitly declared constants as arguments.)

Chapter 15

Conclusion

15.1 Summary of Contributions

Our goal in the research presented here was to investigate the applicability of general-purpose knowledge libraries to the area of action languages. Our main contributions to that end can be listed as follows:

- We developed a theory of explicit definitions in action language $\mathcal{C}+$, and demonstrated that such definitions allow us to represent actions (and fluents) as special cases of other actions (and fluents). This provided us with a theoretical basis for the idea of having libraries of actions and then using them to describe special cases in specific domains. The MAD language was introduced by [Lifschitz and Ren, 2006] as a modular language, with semantics based on our work about explicit definitions.

- We extended this language in several ways, both in the syntactic dimension and in the semantic dimension. The extended semantics not only provides new features but also addresses some shortcomings of the original semantics,

which were identified during the course of our research.

- We designed and implemented a software system that processes action descriptions written in our extended version of MAD. The system turns a modular action description for MAD into a nonmodular description suitable to be given to CCalc. This implementation allowed us to test our formalizations and increase our confidence in the soundness of our modules.

- We developed a library of basic action description modules which were then used to successfully formalize several classic domains from the literature. Using such a library allowed us to abstract out the common aspects of different domains, making the formalizations much simpler.

- We developed a theory of carriers as a new library module and used it to formalize several more domains from the literature. Using the library helped us recognize structural similarities in domains that may seem quite different at first glance: objects as diverse as briefcases, cars, trucks, boats, airplanes, and even humans, could all be represented using the library module `CARRIER`.

- We developed a library module `MOVE_IN_REGION` for representing movement between places which may be part of larger regions.

- We enhanced MAD with the capability to represent numbers and used this new feature to develop library modules about time and resources. These modules were used to enhance existing formalizations or re-write them in more flexible ways.

15.2 Future Work

15.2.1 Specific Topics for Future Work

The following is a list of topics that would improve upon the contents of this dissertation:

- When we rename constants, the current semantics indicates this by introducing an equivalence between the two constants. This is enough for ensuring that the two action occurrences are equivalent — the actions always occur together. However, when one of the actions appears as an argument to a fluent such as `Duration` then the equivalence does not have the effect of making both actions have the same duration. Differentiating between the equivalence of action occurrences and the equivalence of action names would be a very useful extension. (It would also allow us to treat concurrency without using explicit actions.)

- Language $\mathcal{C}+$ has an **unless** clause which can be used to make causal laws defeasible. Ideally, all causal laws in our library should be defeasible, but in order for this to happen, we need to investigate how to make **unless** work in a modular way.

- In some domains we need not only numbers, but also the ability to count things having a certain property. For example, in the missionaries domain, we may want to talk about both individual missionaries and the total number on a bank. For this, we need to have a way to count individuals. Having the ability to count will also enable us to represent domains where numeric resources are affected concurrently (such as multiple purchases increasing the total amount of money in a store) — domains where additive fluents are needed.

- The implemented version of MAD used in this dissertation restricts the right-hand side of a constant renaming statement to be a single constant, prohibiting complex formulas. This prevents us from being able to express certain "conditional renamings" where a renamed constant is redefined in terms of more than one constant in the importing module. The original MAD language allows such complex formulas but they lead to nondefinite axioms that cannot be handled by the current implementation. More work is needed to explore how these axioms may be processed automatically.

- After our implementation processes the modular action description to turn it into a single module in $\mathcal{C}+$, we can use reasoning engines other than CCalc. In particular, answer set programming is sufficiently close to $\mathcal{C}+$ that we believe a practical automated translation will be possible, enabling us to use logic programming systems such as Smodels.

15.2.2 The Big Picture

Knowledge about actions is a small but important part of commonsense knowledge, and the research presented here is a small step towards solving the problem of generality in artificial intelligence. We plan to continue this work by building the library further, though such a library will remain a work-in-progress for a long time, due to the vast amount of commonsense knowledge in the world.

In deciding which modules to include in the library, we will continue to be guided by the large number of small domains that have been represented in $\mathcal{C}+$ by other researchers, as well as domains studied in other work on planning and reasoning about actions. As the library grows, we hope that it will enable us to tackle more small domains with ease and slowly move up to larger and larger

domains.

Another resource to draw upon when building the library will be previous work on hierarchical libraries of components, such as the KM component Library [Barker *et al.*, 2001]. We will benefit from such work in deciding which modules to add to our library and what sorts of axioms these modules should encode. The designers of the KM Component library also take inspiration from linguistic resources such as dictionaries and thesauri to decide which components to encode, to represent general and intuitive concepts useful for knowledge representation. Choosing good names for our components will become more important as the library grows, because it will be crucial in helping users find the right modules for their purposes.

Appendix A

Technical Review of $\mathcal{C}+$

The semantics of $\mathcal{C}+$ is defined in terms of nonmonotonic causal theories [Giunchiglia *et al.*, 2004]. Our review in this chapter introduces nonmonotonic causal theories, followed by the semantics of $\mathcal{C}+$.

A.1 Nonmonotonic Causal Theories

Consider a multi-valued signature as described in Section 3.1.

An *causal theory* is a set of *(causal) rules*—expressions of the form

$$F \Leftarrow G$$

where F and G are formulas.

The *reduct* T^I of a causal theory T relative to an interpretation I is the set of the heads of all rules in T whose bodies are satisfied by I. If I is the unique model of T^I, then it is a *model* of T.

A.2 Semantics of $\mathcal{C}+$

We mentioned in Section 3.3 that every $\mathcal{C}+$ action description represents a transition system—a directed graph whose vertices are states, and whose edges are labeled by events.

The transition system $TS(D)$ represented by an action description D (with signature σ) is defined in terms of an infinite sequence D_0, D_1, ... of causal theories. For any nonnegative integer m, the causal theory D_m is defined as follows.

The signature σ_m of D_m consists of the pairs $i : c$ such that

- $i \in \{0, \ldots, m\}$ and c is a fluent constant of D, or
- $i \in \{0, \ldots, m-1\}$ and c is an action constant of D.

The domain of $i : c$ is the same as the domain of c. The expression $i : F$ denotes the result of inserting $i :$ in front of every occurrence of every constant in a formula F.

Causal theory D_m characterizes histories of length m over action description D. Intuitively, if c is a fluent constant, $i : c$ represents the value of c at time i; if c is an action constant, $i : c$ represents the value of c between times i and $i + 1$.

The rules of D_m are:

$$i : F \Leftarrow i : G$$

for every static law (3.1) in D and every $i \in \{0, \ldots, m\}$, and for every action dynamic law (3.1) in D and every $i \in \{0, \ldots, m-1\}$;

$$i + 1 : F \Leftarrow (i + 1 : G) \wedge (i : H)$$

for every fluent dynamic law (3.2) in D and every $i \in \{0, \ldots, m-1\}$;

$$0 : c = v \Leftarrow 0 : c = v$$

for every simple fluent constant c and every $v \in Dom(c)$.

The transition system $TS(D)$ is completely characterized by the first two members D_0, D_1 of the sequence of causal theories corresponding to D, as follows:

- A state in $TS(D)$ is an interpretation s of the fluent constants such that the corresponding interpretation $0 : s$ of the signature of D_0 is a model of D_0.
- A transition in $TS(D)$ is a triple $\langle s, e, s' \rangle$ where s and s' are interpretations of the fluent constants and e is an interpretation of the action constants, such that the corresponding interpretation $(0 : s) \cup (0 : e) \cup (1 : s')$ of the signature of D_1 is a model of D_1.

Appendix B

Technical Review of MAD

B.1 Syntax of MAD import Statements

Recall from Section 5.4 that the general form of a MAD import statement is

$$\begin{aligned} &\textbf{import } \mathit{NAME};\\ &\quad s_1 \textbf{ is } s_1';\\ &\quad s_k \textbf{ is } s_k';\\ &\quad c_1 \cdots \textbf{ is } F_1;\\ &\quad ...\\ &\quad c_l \cdots \textbf{ is } F_l; \end{aligned} \tag{B.1}$$

where $NAME$ is a module name, $s_1, ..., s_k, s_1', ..., s_k'$ are sort names, $c_1, ...c_l$ are constant names, and $F_1, ..., F_l$ are formulas . The dots after each c_j represent the possibility of having variables as the arguments and domain of constants.

The general syntax of contant renaming **is** clauses in MAD has the form

$$c(x_1, \ldots, x_p) = y \textbf{ is } F$$

where c is a constant, $x_1, \ldots, x_p$ and y are variables (y must be of the same sort as constant c) and F is a formula. If c is a Boolean constant, then "$= y$" may be dropped. It is required that

- the variables to the left of **is** be pairwise distinct,
- every free variable of F occur to the left of **is**,
- the formula

$$\forall x_1 \ldots x_p \exists y' \forall y (F \equiv y = y')$$

be universally valid.

The last condition above expresses that, given specific values for $x_1 \ldots x_p$ (i.e., the constant c has been "grounded"), formula F must be satisfied for exactly one value of y. This condition is similar to the final condition given in Section 4.1 for an explicit definition. That condition stated that, for (4.3) to be an explicit definition of c in terms of σ, exactly one value of c should correspond to any interpretation of σ. Here, the role of σ is played by the constants occurring in F.

B.2 Semantics of MAD

The semantics of MAD is defined by translating MAD into $\mathcal{C}+$. It is composed of two parts: A MAD action description (possibly containing many modules) is first turned into a single-module action description which is considered to have the same meaning. Then this single-module action description is turned into an action description in $\mathcal{C}+$.

Generation a single-module action description

Turning a MAD action description into an equivalent single-module is accomplished by three auxiliary functions:

- $\alpha(M, IS, m)$, takes an import statement IS of the form (B.1) such that $NAME$ is the name of M and turns the module M mentioned in it into a specialized form, by modifying it according to sort and constant renaming clauses in the import statement. (The parameter m is a positive integer, which we will explain below.) It modifies M by:

  - replacing every occurrence of each of the sort names s_i is by $s'_i \quad (i = 1, ..., k)$,
  - prepending "Im." to every occurrence of every variable name and to every occurrence of each of the constant names $c_j \quad (j = 1, ..., l)$,
  - inserting the equivalences

$$\mathrm{I}m.c_j \cdots \equiv F_j, \quad (j = 1, ..., l)$$

    corresponding to the constant renaming clauses from (B.1), at the beginning of the axiom part.

  (The conditions on constant renaming clauses guarantee that the equivalences generated by α are explicit definitions of the constants c_j.)

  An important detail is that all of the variables in a module are renamed so that variable names are always local to a module. Function α is only applicable if the module mentioned contains no import statements itself.

- $\beta(M, M')$, merges one module M with a second module M' that does not

contain import statements. It simply combines the declarations and the axioms, merging repeated declarations. It is undefined if there are mismatched declarations (such as the same identifier declared as a sort in one module and an object in the other).

- $\gamma(M_1; ...; M_n)$, takes a MAD action description $M_1; ...; M_n$ and uses α and β to eliminate the first import statement in the action description. By $\gamma(M_1; ...; M_n)$ we denote the action description obtained by replacing M_i, the first module in $M_1; ...; M_n$ that contains an import statement, with

$$\beta(M, \alpha(M_j, IS, m))$$

where

  - IS is the first import statement in M_i,
  - M is the module obtained from M_i by dropping IS,
  - M_j is the module that IS refers to,
  - m is the smallest positive integer such that the string "Im." does not occur in $M_1; ...; M_n$.

Each application of γ decreases the number of import statements by 1. Repeated application of γ eventually removes all import statements by incorporating imported modules into later modules. The last module in the resulting action description contains the effects of all incorporations and is taken as the single-module action description equivalent to the original description.

Turning a single-module MAD description into a $\mathcal{C}+$ description

Given a single-module action description M we can turn it into a $\mathcal{C}+$ action description using a function U that assigns a finite nonempty set of symbols to each sort name s in M, with the constraint that $U(s)$ must contain all of the objects declared to be of sort s (and no other names declared in M). The resulting $\mathcal{C}+$ action description M_U is defined as follows:

- The signature of M_U contains all the constants from the constants declaration of M, with the variables of sort s replaced by all possible elements of $U(s)$.
- The axioms of M_U are the axioms of M, with the variables of sort s replaced by all possible elements of $U(s)$ and quantifiers are replaced by finite conjunctions or disjunctions.

For example, if we define $U(Latch) = \{L_1, L_2\}$ for the module TWO_LATCHES from Section 5.2, then the corresponding $\mathcal{C}+$ action description TWO_LATCHES$_U$ will represent exactly the transition system in Figure 3.1.

Appendix C

Proofs of Propositions

We begin by proving several lemmas about causal theories, since the semantics of an action description is defined in terms of a causal theory. These lemmas are then used in the proofs of Propositions 1–5.

C.1 Some Properties of Causal Theories

The following is a counterpart of Proposition 2 for causal theories. It is a restatement of Proposition 1 from [Turner, 2004].

Lemma 1 *Let T be a causal theory containing a rule of the form*

$$F \equiv G \Leftarrow \top.$$

The causal theory obtained by replacing an occurrence of F by G in any other rule of T has the same models as T.

Since Propositions 2 and 3 are about explicit definitions in $\mathcal{C}+$, we define the counterpart of this concept for causal theories. An *explicit definition* of a multi-

valued constant c, *in terms of* a multi-valued signature σ which does not contain c, is a set of causal laws of the form

$$c = v \equiv F_v \ \Leftarrow \top, \tag{C.1}$$

one for each $v \in Dom(c)$, where

- each F_v is a formula of σ, and
- the formulas

  $$\bigvee_{v \in Dom(c)} F_v$$

  and

  $$\bigwedge_{v,w \in Dom(c), v \neq w} \neg(F_v \wedge F_w)$$

  are tautological.

The following is a counterpart of Proposition 2 for causal theories.

Lemma 2 *Let T be a causal theory of a signature σ, and let c be a constant that does not belong to σ. If T' is a causal theory of the signature $\sigma \cup \{c\}$ obtained from T by adding an explicit definition of c in terms of σ, then $X \mapsto X|_\sigma$ is a 1-1 correspondence between the models of T' and the models of T.*

Proof: Let the set of formulas in the heads of rules (C.1) be called C. The task of proving that $X \mapsto X|_\sigma$ is a 1-1 correspondence between the models of T' and the models of T can be divided into three parts.

Part I: Showing that if X is a model of T' then $X|_\sigma$ is a model of T.

Assume that X is a model of T'. Then $X \models (T')^X$, and consequently $X \models T^X$. Since T doesn't contain c, $X \models T^{X|_\sigma}$. Since $T^{X|_\sigma}$ doesn't contain c, it follows

that $X|_\sigma \models T^{X|_\sigma}$.

We also need to show that $X|_\sigma$ is the unique model of $T^{X|_\sigma}$. Let Y be any model of $T^{X|_\sigma}$. Define a new interpretation Y' of the signature $\sigma \cup \{c\}$ such that $Y'|_\sigma = Y$ and $Y'(c)$ is $v \in Dom(c)$ for which Y satisfies F_v. (Under the assumptions of the theorem, such a v is unique.) Clearly, Y' satisfies $T^{X|_\sigma}$ because Y does. Also, Y' was defined in a way that ensures it satisfies C. Therefore Y' is a model of $(T^{X|_\sigma} \cup C) = (T')^X$. Since X is the unique model of $(T')^X$, it follows that $Y' = X$, and $Y = Y'|_\sigma = X|_\sigma$. Thus $X|_\sigma$ is a model of T.

Part II: Showing that every model of T can be represented in the form $X|_\sigma$, where X is a model of T'.

Let Y be a model of T. Define a new interpretation X of the signature $\sigma \cup \{c\}$ such that $X|_\sigma = Y$ and $X(c)$ is $v \in Dom(c)$ for which Y satisfies F_v. Clearly, X satisfies T^Y because Y does. Also, X is defined to satisfy C. Therefore X is a model of $(T^Y \cup C) = (T')^X$. Now we need to show that X is the unique model of $(T')^X$.

Let X' be any model of $(T')^X$. Then X' satisfies $(T')^X = T^Y \cup C$. Since T doesn't contain c, $X'|_\sigma$ satisfies T^Y. But Y is the unique model of T^Y so $X'|_\sigma = Y = X|_\sigma$. Since X' satisfies C,

$$X' \models c = X'(c) \equiv F_{X'(c)}$$

so that $X' \models F_{X'(c)}$. Then $Y \models F_{X'(c)}$. By the choice of $X(c)$, $Y \models F_{X(c)}$. It follows that $X(c) = X'(c)$. We have shown that $X' = X$, so X is the unique model of $(T')^X$, and therefore a model of T', such that $X|_\sigma = Y$.

Part III: Showing that no model of T can be represented in the form $X|_\sigma$, where X is a model of T', in more than one way.

Let X and Z be models of T' such that $X|_\sigma = Z|_\sigma$. Then X is the unique model of $(T')^X$ and Z is the unique model of $(T')^Z$. Since the bodies of rules in T' don't contain c, $(T')^Z = (T')^X$. So Z is the unique model of $(T')^X$. Therefore $Z = X$.

The following is a counterpart of Proposition 3 for causal theories.

Lemma 3 *Let σ be a signature and c be a constant that does not belong to σ. Let T be a causal theory of signature $\sigma \cup \{c\}$ which does not contain c in the heads of rules. Let T' be a causal theory of signature $\sigma \cup \{c\}$ obtained from T by adding an explicit definition (C.1) of c in terms of σ. Then the causal theory T'' of signature $\sigma \cup \{c\}$ obtained from T by adding the rules*

$$c = v \Leftarrow F_v \qquad (v \in Dom(c))$$

has the same models as T'.

Proof: *Left to right:* Let X be a model for T'. Then X is the unique model of $T^X \cup (T' \setminus T)^X$. Since X is a model of $(T' \setminus T)^X$, $X \models F_{X(c)}$ and, for all $w \neq X(c)$, $X \not\models F_w$. Therefore, $(T'' \setminus T)^X = \{c = X(c)\}$ and X is a model for $(T'' \setminus T)^X$. Consequently, X is a model for $(T'')^X$. We need to show that it is the unique model.

Let $Y \models (T'')^X$. Since $(T'' \setminus T)^X = \{c = X(c)\}$, $Y(c) = X(c)$. Since (C.1) is an explicit definition, for some $w \in Dom(c)$, $Y \models F_w$ and, for all $x \neq w$, $Y \not\models F_x$. Take the interpretation Y' of $\sigma \cup \{c\}$ such that $Y'|_\sigma = Y|_\sigma$ and $Y'(c) = w$. Then $Y' \models (T' \setminus T)^X$ and $Y' \models T^X$ (since $Y \models T^X$ and T^X doesn't contain c). By the fact that X is the unique model of $(T')^X$, $Y' = X$. So $Y|_\sigma = Y'|_\sigma = X|_\sigma$. Consequently, $Y = X$.

Right to left: Let X be a model for T''. Then X is the unique model of $T^X \cup (T'' \setminus T)^X$. Since (C.1) is an explicit definition, $(T'' \setminus T)^X$ is a singleton, $\{c = X(c)\}$. Consequently $X \models F_{X(c)}$ and for all $w \neq X(c)$, $X \not\models F_w$. Then $X \models (T' \setminus T)^X$ and X is a model for $(T')^X$. We need to show that it is the unique model.

Let $Y \models (T')^X$. Take the interpretation Y' of $\sigma \cup \{c\}$ such that $Y'|_\sigma = Y|_\sigma$ and $Y'(c) = X(c)$. Then $Y' \models T^X$ (since $Y \models T^X$ and T^X doesn't contain c) and $Y' \models (T'' \setminus T)^X$. Since X is the unique model of $(T'')^X$, $Y' = X$ so $Y|_\sigma = Y'|_\sigma = X|_\sigma$. Since X satisfies only $F_{X(c)}$ among formulas F_v ($v \in Dom(c)$) and these formulas don't contain c, Y also satisfies only $F_{X(c)}$. Since $Y \models (T' \setminus T)^X$, $Y(c) = X(c)$.

To prove Proposition 3 we will also need the following modification of Lemma 3.

Lemma 4 *Let σ be a signature and c be a constant that does not belong to σ. Let T be a causal theory of signature $\sigma \cup \{c\}$ which does not contain c in the heads of rules. Let T' be a causal theory of signature $\sigma \cup \{c\}$ obtained from T by adding an explicit definition (C.1) of c in terms of σ and the rules*

$$c = v \quad \Leftarrow \quad c = v \tag{C.2}$$

$$F_v \quad \Leftarrow \quad F_v \tag{C.3}$$

for all v from $Dom(c)$. Then the causal theory T'' of signature $\sigma \cup \{c\}$ obtained from T by adding rules (C.2), (C.3) and

$$c = v \Leftarrow F_v \qquad (v \in Dom(c))$$

has the same models as T'.

The proof is very similar to that of Lemma 3. (Instead of constructing Y', we can simply use Y.)

The other lemmas that we need are related to the concept of strong equivalence, which was originally introduced for logic programs in [Lifschitz *et al.*, 2001] and was extended to causal theories in [Turner, 2004]; also see [Sergot and Craven, 2005].

Causal theories T_1 and T_2 of the same signature σ are *equivalent* if they have the same models. They are *strongly equivalent* if, for every causal theory T of a signature σ' containing σ, the theories $T_1 \cup T$ and $T_2 \cup T$ of the signature σ' are equivalent.

Lemma 5 *Let T_1 and T_2 be causal theories with a common signature, such that for any interpretation J of their signature, T_1^J is equivalent to T_2^J. Then T_1 and T_2 are strongly equivalent.*

This lemma is slightly weaker than Theorem 1 from [Turner, 2004], which gives a complete characterization of strong equivalence in terms of pairs of interpretations.

Proof: Let T be a causal theory. For any interpretation J of the signature of T,

$$
\begin{array}{ll}
 & J \text{ is a model of } T_1 \cup T \\
\text{iff} & J \text{ is the unique model of } T_1^J \cup T^J \\
\text{iff} & J \text{ is a model of } T_1^J \cup T^J \\
 & \quad \text{and for any model } I \text{ of } T_1^J \cup T^J, I = J \\
\text{iff} & J \text{ is a model of } T_2^J \cup T^J \\
 & \quad \text{and for any model } I \text{ of } T_2^J \cup T^J, I = J
\end{array}
$$

iff J is the unique model of $T_2^J \cup T^J$

iff J is a model of $T_2 \cup T$.

Lemma 6 *Let F be a formula of the signature σ. The causal theory T consisting of rules of the form*

$$a \Leftarrow a$$

for all atoms a of σ, is strongly equivalent to the theory obtained from T by adding the rule

$$F \Leftarrow F.$$

Proof: Call the second theory T'. By Lemma 5, all we need to check is that, for any interpretation J of T, T^J and $(T')^J$ are equivalent. Note that due to the form of the rules in T, for any interpretation J of σ, T^J is satisfied only by J. Similarly, $(T')^J$ is also satisfied by J only.

The following is a counterpart of Proposition 5 for causal theories.

Lemma 7 *Let F be a formula of a signature σ and G be a formula of a signature σ' containing σ. Let T be the causal theory of consisting of the rules*

$$a \Leftarrow a$$

for all atoms a of σ. Then

$$T$$

$$F \Leftarrow G$$

is strongly equivalent to

$$T$$

$$\bot \Leftarrow \neg F \wedge G.$$

Proof: Call the first theory T_1, the second T_2. By Lemma 5, all we need to check is that, for any interpretation J of σ', T_1^J and T_2^J are equivalent. Due to the form of the rules in T, for any interpretation I of σ', if I satisfies T^J, then $I|_\sigma = J|_\sigma$ and consequently $I \models F$ iff $J \models F$. It follows that

$$\begin{aligned}
& I \models T_1^J \\
\text{iff} \quad & I \models T^J \text{ and } (I \models F \text{ if } J \models G) \\
\text{iff} \quad & I \models T^J \text{ and } (J \models F \text{ if } J \models G) \\
\text{iff} \quad & I \models T^J \text{ and } J \models G \supset F \\
\text{iff} \quad & I \models T^J \text{ and } J \not\models \neg F \wedge G \\
\text{iff} \quad & I \models T_2^J.
\end{aligned}$$

C.2 Proofs of Propositions 1–5

Proposition 1 *Let F, G be formulas, let D be an action description, and let L, L' be similar causal laws such that L' is obtained from L by replacing an occurrence of F by G. Then the action description*

$$D$$

L

caused $F \equiv G$

represents the same transition system as

D

L'

caused $F \equiv G$.

Proof: Let the first action description be A and the second A'. Since the transition systems $TS(A)$ and $TS(A')$ are characterized by A_0, A_1 and A'_0, A'_1, it suffices to show that A_m has the same models as A'_m. First, note that, due to its form, the last law in A and A' above must be an action dynamic law or a static law. The causal theories A_m and A'_m will contain rules

$$i : F \equiv i : G \Leftarrow \top$$

where i ranges over $\{0, \ldots, m-1\}$ or $\{0, \ldots, m\}$, depending on whether the causal law is an action dynamic law or a static law. Theory A'_m can be obtained from A_m by replacing some formulas of the form $i : F$ by $i : G$. Therefore, by Lemma 1, the theories A_m and A'_m have the same models.

Proposition 2 *Let D be an action description of a signature σ, and let c be a constant that does not belong to σ. If D' is an action description of the signature $\sigma \cup \{c\}$ obtained from D by adding an explicit definition of c in terms of σ, then D*

is a residue of D'.

Proof: Action description D is a residue of D' if the mapping

$$s \mapsto s|_\sigma \tag{C.4}$$

is a 1-1 correspondence between the states of $TS(D')$ and the states of $TS(D)$, and the mapping

$$\langle s_0, e, s_1\rangle \mapsto \langle s_0|_\sigma, e|_\sigma, s_1|_\sigma\rangle \tag{C.5}$$

is a 1-1 correspondence between the transitions of $TS(D')$ and the transitions of $TS(D)$.

Since the semantics of an action description D is characterized in terms of the corresponding causal theories D_0 and D_1, to prove that (C.4) is a 1-1 correspondence between the states of $TS(D')$ and the states of $TS(D)$ we need to check that

$$0{:}s \mapsto 0{:}(s|_\sigma) \tag{C.6}$$

is a 1-1 correspondence between the models of D_0' and the models of D_0, and to prove that (C.5) is a 1-1 correspondence between the transitions of $TS(D')$ and the transitions of $TS(D)$ we need to check that

$$0{:}s_0 \cup 0{:}e \cup 1{:}s_1 \mapsto 0{:}(s_0|_\sigma) \cup 0{:}(e|_\sigma) \cup 1{:}(s_1|_\sigma) \tag{C.7}$$

is a 1-1 correspondence between the models of D_1' and the models of D_1.

First consider the case when the explicitly-defined constant c is a simple fluent constant. For every simple fluent constant d from $\sigma \cup \{c\}$, by $R(d)$ we denote

the set of rules

$$0 : d = v \Leftarrow 0 : d = v$$

for all $v \in Dom(d)$. Clearly D'_m is obtained from D_m by adding the rules

$$i : c = v \equiv i : F_v \Leftarrow \top \qquad (v \in Dom(c)) \tag{C.8}$$

where i ranges over $\{0, \ldots, m\}$ and the rules

$$0 : c = v \Leftarrow 0 : c = v \qquad (v \in Dom(c)), \tag{C.9}$$

that is, $R(c)$. Note first that dropping the rules $R(c)$ from D'_m does not change the set of models. Indeed, according to Lemma 1, in the presence of (C.8) the rules $R(c)$ can be replaced by

$$0 : F_v \Leftarrow 0 : F_v \qquad (v \in Dom(c)). \tag{C.10}$$

Since all constants occurring in F_v are simple fluent constants from σ, D_m contains the rules $R(d)$ for all such constants d. By Lemma 6, in the presence of these rules (C.10) can be dropped.

To conclude the proof for the case when c is a simple fluent constant, it remains to observe that rules (C.8) can be viewed as an explicit definition of $i : c$ in terms of $i : \sigma$. By Lemma 2, (C.6) is a 1-1 correspondence between the models of D_0 and the models of D'_0, and (C.7) is a 1-1 correspondence between the models of D_1 and the models of D'_1.

When the constant c is a statically determined fluent constant or an action constant, the proof is similar but simpler, since there are no rules (C.9), so we don't need to use Lemma 6.

Proposition 3 *Let σ be a signature and c be a constant that does not belong to σ. Let D be an action description of signature $\sigma \cup \{c\}$ which does not contain c in the heads of laws. Let D' be an action description of signature $\sigma \cup \{c\}$ obtained from D by adding an explicit definition (4.3) of c in terms of σ. Then the action description of signature $\sigma \cup \{c\}$ obtained from D by adding the rules*

$$\textbf{caused } c = v \textbf{ if } F_v \qquad (v \in Dom(c))$$

represents the same transition system as D'.

Proof: First consider the case when the explicitly-defined constant c is a simple fluent constant. Call the second action description D''. The difference between D'_m and D''_m is that the former includes rules

$$i : c = v \equiv i : F_v \Leftarrow \top \qquad (v \in Dom(c)) \tag{C.11}$$

whereas the latter includes

$$i : c = v \Leftarrow i : F_v \qquad (v \in Dom(c)), \tag{C.12}$$

where i ranges over $\{0, \ldots, m\}$. In addition to these, both contain D_m and the rules

$$0 : c = v \Leftarrow 0 : c = v \qquad (v \in Dom(c)) \tag{C.13}$$

because c is a simple fluent constant. Since F_v contains only simple fluent constants in σ, causal theories D'_m and D''_m will contain rules of the same form as (C.13) for

any fluent constants in $0 : F_v$. Therefore, using Lemma 6, we may add the rules

$$0 : F_v \Leftarrow 0 : F_v \qquad (v \in Dom(c)) \tag{C.14}$$

to D'_m and D''_m without changing their models.

By one application of Lemma 4 and $m - 1$ applications of Lemma 3, causal theories D'_m and D''_m have the same models.

When the constant c is a statically determined fluent constant or an action constant, the proof is similar but simpler, since there are no rules (C.13), so we don't need to use Lemma 6 or Lemma 4.

Proposition 4 *MB is a residue of MB*.*

The proof uses the concept of strong equivalence of action descriptions. Two action descriptions D and D' of the same signature are *equivalent* if $TS(D) = TS(D')$. They are *strongly equivalent* if for any action description D'' (of a possibly larger signature), action descriptions $D \cup D''$ and $D' \cup D''$ are equivalent. A similar definition appears in Section 5 of [Sergot and Craven, 2005].

The lemma below refers to the following explicit definitions of constants in (4.12).

$$\textbf{caused } Location(p) = l \equiv Loc(p) = l \tag{C.15}$$

$$\textbf{caused } Move(Box) = true \equiv \bigvee_{l_0 \in L} PushBox(l_0) \tag{C.16}$$

$$\textbf{caused } Move(Box) = false \equiv \neg \bigvee_{l_0 \in L} PushBox(l_0) \tag{C.17}$$

$$\textbf{caused } Move(p) = true \equiv \bot \qquad (p \neq Box) \tag{C.18}$$

caused $Move(p) = false \equiv \top \qquad (p \neq Box)$ (C.19)

caused $Mover(Box) = Monkey \equiv \bigvee_{l_0 \in L} PushBox(l_0)$ (C.20)

caused $Mover(Box) = None \equiv \neg \bigvee_{l_0 \in L} PushBox(l_0)$ (C.21)

caused $Mover(Box) = Bananas \equiv \bot$ (C.22)

caused $Mover(Box) = Box \equiv \bot$ (C.23)

caused $Mover(p) = None \equiv \top \qquad (p \neq Box)$ (C.24)

caused $Mover(p) = Monkey \equiv \bot \qquad (p \neq Box)$ (C.25)

caused $Mover(p) = Bananas \equiv \bot \qquad (p \neq Box)$ (C.26)

caused $Mover(p) = Box \equiv \bot \qquad (p \neq Box)$ (C.27)

caused $Destination(Box) = l \equiv PushBox(l) \wedge \bigwedge_{\substack{l_0 \in L\} \\ l_0 < l}} \neg PushBox(l_0)$ (C.28)

caused $Destination(Box) = None \equiv \neg \bigvee_{l_0 \in L} PushBox(l_0)$ (C.29)

caused $Destination(p) = l \equiv \bot \qquad (p \neq Box)$ (C.30)

caused $Destination(p) = None \equiv \top \qquad (p \neq Box)$ (C.31)

(In formulas (C.28) and (C.29), L stands for $\{L_1, L_2, L_3\}$ and the relation $<$ on this set is defined by $L_1 < L_2 < L_3$.)

Lemma 8 *MB^* is strongly equivalent to the action description which consists of MB and (C.15)–(C.31).*

The proof of this lemma is given by a long series of strongly equivalent transformations. We do not include them here.

Proposition 4 may be derived from the lemma by applying Proposition 2 to

each of the constants in (4.12).

Proposition 5 *Let D be an action description and F be a formula such that all constants in F are action constants which are exogenous in D. Then*

$$D$$
$$\textbf{caused } F \textbf{ if } G$$

represents the same transition system as

$$D$$
$$\textbf{caused } \bot \textbf{ after } \neg F \wedge G.$$

Proof: Let the first action description be A and the second A'. Let σ be the set of constants occurring in F. Causal theory A_m will contain

$$i : F \Leftarrow i : G$$

and A'_m will contain

$$\bot \Leftarrow \neg i : F \wedge i : G$$

where i ranges over $\{0, \ldots, m-1\}$. Due to the requirement that constants in F are action constants which are exogenous, both A_m and A'_m will contain rules of the form

$$i : a \Leftarrow i : a$$

for all atoms a of σ, where i ranges over $\{0, \ldots, m-1\}$. By Lemma 7, the theories

A_m and A'_m have the same models.

Appendix D

Input Language of the MAD Implementation

A MAD input file contains an action description, along with comments.

D.1 Comments

A comment in the input begins with the character `%` and lasts until the end of the line (i.e. until a newline character). Everything in the comment is ignored by the parser.

D.2 Include Statements

An input file may contain one or more "include" statements before the action description. These statements specify (within quotes) names of other files containing action descriptions. They are treated as if the contents of the included files appear in place of the include statements. For example, to include a file named `library`

(located in the same directory) before an action description, we can write the following:

```
include "library"
```
action_description_which_may_refer_to_modules_from_included_file

If the filename specifies just a simple filename, the file is expected to be in the directory in which the program is executed. If a relative path to a file is given, this is interpreted as relative to the directory in which the program is executed.

If the included file does not contain a complete action description, an error will occur.

Include statements can only appear at the start of a file, and the final include statement in a file must be followed by an action description.

Include statements may be nested in the sense that a file which is referred to in an include statement may contain an include statement itself.

D.3 Identifiers and Keywords

Identifiers must begin with a letter. The letter may be followed by a combination of letters, numbers, or "_". In `grep`-like regular expression notation, we may write this as

```
Identifier:        [a-zA-Z][a-zA-Z0-9_]*
```

(Note for those familiar with the semantics of MAD: this notation allows I*integer* to appear in an identifier, though any appearance of a "." will cause a syntax error.)

The following words are keywords of the language and may not appear as identifiers:

`include, module, import, is, case, numeric_symbol, sorts, inclusions, objects, actions, fluents, variables, action, explicitAction, simple, staticallyDetermined, rigid, axioms, if, after, constraint, default, exogenous, causes, nonexecutable, always, may, cause, inertial, exists, forall, true, false, Boolean`

D.4 Action Descriptions

An action description is composed of a series of four basic components:

- numeric symbol declarations
- sort declarations,
- inclusion declarations,
- modules,

which may be interleaved. These components are described below.

D.5 Numeric Symbol Declarations

A sort declaration is of the form

`numeric_symbol` *id*=*int*

where *id* is an identifier and *int* is an integer.

The purpose of such declarations is to have symbolic names for integers. Any occurrence of *id* occurring after the above numeric symbol declaration will be treated as an occurrence of *int*.

Identifiers declared as numeric symbols may only occur in places where integers are allowed.

D.6 Sort Declarations

A sort declaration is of the form

`sorts`

s_1 ;

⋮

s_n ;

where $1 \leq n$, and each s_i is an identifier or the keyword `Boolean`.

A sort may not be declared more than once.

D.6.1 Built-in Sorts: `Boolean`, and Integer Ranges "$m..n$"

The keyword `Boolean` is a built-in sort name, declared implicitly. (Later, in the section about objects, we will see that this built-in sort contains two built-in objects, `true` and `false`.)

An expression of the form "$m..n$", where m and n are integers, represents the set of integers between m and n inclusively. Such integer ranges are built-in sorts in MAD (with the integers as built-in objects belonging to those sorts) and they are considered to be declared implicitly.

Declaring `Boolean` is allowed but a warning is given. Declaring integer ranges as sorts is prohibited.

Definition: We call the keyword `Boolean`, integer ranges of the form "m`..`n", and sort names s_i declared as above *simple sort names*.

D.7 Inclusion Declarations

An inclusion declaration is of the form

`inclusions`

i_1 `;`

$\vdots$

i_n `;`

where $0 \leq n$, and each i_i is an inclusion expression, as defined below.

An inclusion expression is of the form

s_1 `<<` s_2 `<<` ... `<<` s_n

where $(n > 1)$, s_1 is a simple sort name other than an integer range, and s_2, ..., s_n are sort names other than `Boolean`.

The reason that the keyword `Boolean` or an integer range cannot appear as a supersort (on the right of `<<`) is that such an appearance would lead to these built-in sorts having objects other than their built-in objects.

We may think of the sort and inclusion declarations as forming a forest, with the sorts as vertices and each `<<` relation standing for an edge from the right-hand

sort to the left-hand sort.

Inclusions leading to cycles in the sort inclusion forest are prohibited. (Attempting to make such declarations will cause the system to give an error message.)

D.8 Modules

A MAD module is of the form

`module` *module-name* `;`

module-body

where *module-name* is an identifier. The *module-body* consists of an ordered series of sections, any of which may appear at most once. These are, in order, object declarations, action constant declarations, fluent constant declarations, variable declarations, and axioms. In addition, the *module-body* may contain any number of import declarations in before or after (but not within) these sections.

Each module in an action description must have a unique name.

D.9 Object Declarations

An object declaration is of the form

`objects`

o_spec_1 `;`

$\vdots$

o_spec_n ;

where $0 \leq n$, and each o_spec_i is an object specification, as defined below.

An object specification is of the form

$o_1, \ldots, o_m$: *sort*

where $1 \leq m$, each o_i is an identifier, possibly followed by a parenthesized list of arguments, and *sort* is a sort name other than `Boolean`. Any arguments following o_i must be a simple sort name.

Example. Here is an example object declaration, assuming sorts `Person`, and `Object` have been declared:

```
objects
Player(1..10) :  Person;
Hat(Person) :  Object;
```

The keywords `true` and `false` are built-in objects, of sort `Boolean`, declared implicitly in every module. These objects cannot be declared explicitly. Also, no other objects may be declared to be of sort `Boolean`.

Integers are also built-in objects, and every integer is considered to be declared implicitly, so that it can be used in formulas, as will be seen below.

We may think of the sort declarations, inclusion declarations, and object declarations as forming a "sort dependency" forest. The vertices are sorts. Each object o_i specified above adds an edge for every argument it has. The edge points

from *sort* to the argument. Inclusions of the form "s_1 `<<` s_2" add edges from s_2 to all of the sorts which s_1 already points to.

Example. The example object declaration above adds the edges

$$\texttt{Person} \rightarrow 1..10$$

$$\texttt{Object} \rightarrow \texttt{Person}$$

to the sort dependency forest.

Cycles in the sort dependency forest are prohibited. (Attempting to make any declarations which result in a cycle will cause the system to give an error message.)

An object may not be declared more than once, except when the re-declaration is part of an imported module. (See the section on import declarations below.) The re-declaration must match the original declaration, i.e., the two declarations must assign the same sort to the object.

D.10 Action Constant Declarations

An action constant declaration is of the form

`actions`

c_1 `;`

$\vdots$

c_n `;`

where $1 \leq n$, and each c_i is an identifier, possibly followed by a parenthesized

list of arguments. An action constant argument is a simple sort name. No domain is specified when declaring an action because all actions are `Boolean`-valued.

Example. Here is an example action constant declaration, assuming sorts `row`, `column`, and `color` have been declared:

```
actions
Paint_square(row, column, color);
```

The arguments of an action constant being declared may not contain the keyword `action`. (Otherwise, imagine two actions *a(*`action`*)* and *b*. When we ground, we would get *b*, *a(b)*, *a(a(b))*, etc., never ending.)

An action constant may not be declared more than once, except when the re-declaration is part of an imported module. (See the section on import declarations below.) The re-declaration must match the original declaration, i.e., the two declarations must assign the same arguments.

D.11 Fluent Constant Declarations

A fluent constant declaration is of the form

`fluents`

c_spec_1 ;

⋮

c_spec_n ;

where $1 \leq n$, and each *c_spec*$_i$ is a fluent constant specification, as defined below.

A fluent constant specification is of the form

$c_1, \ldots, c_m$ `:` *kind*`(`*domain*`)`

where $1 \leq m$, and each c_i is an identifier, possibly followed by a parenthesized list of arguments. A fluent constant argument is a simple sort name or the keyword `action`. The *kind* may be one of the keywords `simple`, `staticallyDetermined`, or `rigid`, indicating, respectively, a simple fluent, statically determined fluent or rigid fluent. The *domain* is a simple sort name.

When declaring simple fluents, statically determined fluents or rigid constants, it is also possible to omit the parenthesized domain, in which case it is implicitly assumed to be `(Boolean)`.

Example. Here are some example fluent constant declarations assuming sorts `row`, `column`, and `color` have been declared:

```
fluents
Game_started: rigid;
Square_color(row, column) : simple(color);
```

A fluent constant may not be declared more than once, except when the re-declaration is part of an imported module. (See the section on import declarations below.) The re-declaration must match the original declaration, i.e., the two

declarations must assign the same arguments, the same kind and the same domain to the fluent constant.

D.12 Variable Declarations

A variable declaration is of the form

`variables`

v_spec_1 ;

$\vdots$

v_spec_n ;

where $0 \leq n$, and each v_spec_i is a variable specification, as defined below.

A variable specification is of the form

$v_1, \ldots, v_m$: *sort*

where $1 \leq m$, each v_i is an identifier, and *sort* is a simple sort name, the keyword `action` or the keyword `explicitAction`.

A variable may not be declared more than once in the same module.

Each variable is local to the module in which it is declared. To use the same identifier as a variable in another module, one has to declare it anew in the latter module (and may declare it as belonging to a different sort).

Variables for actions are different from other variables, because in some contexts they are treated as the action atom and in others as the name of the action. For example, given two action variables `a` and `a1`, the axiom

```
nonexecutable a & a1 if a!=a1
```

may be used to express that two actions are not executable concurrently unless they are the same action. In the first two occurrences, the variables are treated as the action atoms and the latter occurrences on either side of the inequality sign are treated as the names of the actions.

D.13 Axioms

An axioms section is of the form

`axioms`

$axiom_1$;

$\vdots$

$axiom_n$;

where $0 \leq n$, and each $axiom_i$ is an axiom, as defined below.

The axiom section of a MAD action description is different from the declaration sections we described above, in that the previous sections serve to *declare* identifiers whereas this section *uses* these identifiers. From now on, whenever we refer to a constant/object/variable, we mean an identifier which was declared as such in a prior section of the action description.

Axioms in MAD are like causal laws in $\mathcal{C}+$. They are composed of *formulas* and certain keywords such as `caused`, `if`, `after`, etc. We will show the exact forms of acceptable axioms below, but first we need to define what we mean by a valid formula. Formulas are defined recursively, using *terms* and logical connectives. Therefore, to describe what a valid formula is, we need to first define what a term

is.

D.13.1 Terms

A *term* may be

- a constant (followed by a parenthesized list of arguments if so declared)
- a variable
- an object (followed by a parenthesized list of arguments if so declared)
- an integer
- *term* `+` *term*
- *term* `*` *term*

The parenthesized list of arguments following a constant/object must match the arguments listed in the declaration of the constant/object, i,e. the arguments in the constant/object term must be objects/variables of the sort declared for the corresponding argument, or a action constant if the corresponding argument was declared as an action.

Since a constant declaration allows only simple sort names or `action` as arguments, and object declarations only allow simple sort names as arguments, we would normally expect no non-action constants to be allowed as arguments of a constant/object term. However, the system allows such arguments as shorthand for the value the constant has (at that time instant). In other words, an argument for a constant/object term may be a constant term, provided that the latter's domain matches the sort of the former's argument declaration. Formulas with such short-

hand notation are expanded to a certain longhand form. The exact details will be given after formulas are defined.

The last two items above, expressions for arithmetic operations (addition and multiplication) may not appear as arguments to constants/objects. Furthermore, each of the terms in these expressions must be "numerical," meaning an integer, a constant with an integer range domain, a variable with an integer range sort, or another validly formed arithmetic expression.

D.13.2 Formulas

A formula is built from terms and connectives in one of the following ways:

1. zero-place connective `true`
2. zero-place connective `false`
3. a Boolean constant term
4. an action variable
5. *sort-name* `(` *variable* `)`
6. `Boolean (` *variable* `)`
7. `action (` *variable* `)`
8. *term* `=` *term*
9. *term* `!=` *term*
10. *term* `<` *term*
11. `(` *formula* `)`

12. `-`*formula*

13. *formula* `&` *formula*

14. *formula* `|` *formula*

15. *formula* `->` *formula*

16. *formula* `<->` *formula*

17. `exists` *variable formula*

18. `forall` *variable formula*

Items (12) through (16) correspond to the usual logical connectives: (in descending order of precendence) negation, conjunction, disjunction, implication and equivalence. All of the binary connectives are left-associative.

Quantifiers `exists` and `forall` have lower precedence than the logical connectives, so, for example, assuming Q_1, Q_2 are quantifiers, v_1, v_2 are variables, and F, G are formulas,

$$Q_1 v_1 \ F \text{ connective } Q_2 v_2 \ G$$

will be parsed as

$$Q_1 v_1 \ (F \text{ connective } Q_2 v_2 \ G)$$

and not as

$$(Q_1 v_1 \ F) \text{ connective } \ (Q_2 v_2 \ G)$$

Items (5) is shorthand for a quantified formula

`exists` *new-variable-of-sort* *variable=new-variable-of-sort*

and is called a "sort name formula."[1] Items (6) and (7) are similar — they are sort name formulas for built-in sort `Boolean` and for actions.

In items (8) and (9), there are some additional constraints that are checked:

- if one of the terms is an action variable or an explicit action variable, then the other term must be an action variable or an explicit action variable or a Boolean action constant.
- if both of the terms are constants, they must have the same domain
- if one of the terms is a constant and the other is an object/variable, the sort of the object/variable must be the same as the domain of the constant.

In item (10), each term must be numerical, meaning an integer, a constant with an integer range domain, a variable with an integer range sort, or an arithmetic operator term as given in the definition of a term (addition and multiplication).

As mentioned above in the description of terms, MAD also allows having constants as arguments to constant terms, even where a non-action sort is expected. This is shorthand for the value of that constant at the time. For example,

```
Square_color(row_of(queen), column)
```

is valid shorthand if the domain of `row_of` matches the first argument declaration of `Square_color`. This would be expanded by using a new variable, say `r`, from the domain of `row_of`. The minimal formula F in which this term appears would be expanded to

F' `& row_of(queen)=r`

[1] A sort name formula like (5) corresponds to the English sentence "*variable* is of sort *sort-name*."

where F' is obtained by replacing that occurrence of `row_of(queen)` by `r` in F. Note that such shorthands may be nested too.

IMPORTANT: This shorthand is only allowed in formulas appearing as parts of axioms, not in terms appearing as parts of import declarations. (We will describe import declarations below.)

D.13.3 Axioms

An axiom is built from formulas and terms in one of the following ways: (The parts within square brackets are optional)

1. *formula* [`if` *formula*] [`after` *formula*]
2. *formula* `causes` *formula* [`if` *formula*]
3. `default` *formula* [`if` *formula*] [`after` *formula*]
4. `exogenous` *constant* [`if` *formula*]
5. `inertial` *fluent-constant* [`if` *formula*]
6. `constraint` *formula* [`after` *formula*]
7. `nonexecutable` *formula* [`if` *formula*]
8. `always` *formula*
9. `rigid` *fluent-constant*
10. *formula* `may cause` *formula* [`if` *formula*]

Recall, from the paper "Nonmonotonic Causal Theories"[Giunchiglia *et al.*, 2004], that

- an *action formula* is a formula with no fluent constants and at least one action constant,
- a *fluent formula* is a formula with no action constants.

In items (2) and (10) the first formula has to be an action formula and the second should be an action formula or a fluent formula.

In item (7) the first formula has to be an action formula.

All of the axioms listed above may be seen as special cases of item (1).[2] Let us rewrite this general form as

$$F\ [\texttt{if}\ G]\ [\texttt{after}\ H]$$

where F,G and H are formulas. An axiom of this form must satisfy the following conditions in order to be valid:

- F must be such that the axiom is definite, i.e. F is either
  - the zero-place connective false
  - a single atomic formula[3] with at most one constant
  - the negation of an atomic formula with exactly one Boolean constant
- if there is no H part and F is a fluent formula, then G must be a fluent formula
- if there is an H part,
  - F and G must be fluent formulas

[2]The details of how items (2)-(10) may be seen as abbreviations of (1) can be found in Appendix B of [Giunchiglia *et al.*, 2004].

[3]Items (1)-(4) and (7) in the description of formulas above.

– F must not contain any statically determined fluents or rigid constants.

- if F contains a rigid constant, the axiom must not contain any non-rigid constants.

The axioms listed in this section cover all of the abbreviations from Appendix B of [Giunchiglia *et al.*, 2004], with the exception of abbreviations (15)-(17) there, which involve the `unless` construct.

D.14 Import Declarations

An import declaration is of the form

`import` *module-name* ;

sort-renaming-clause$_1$;

⋮

sort-renaming-clause$_n$;

constant-renaming-clause$_1$;

⋮

constant-renaming-clause$_m$;

where each $(n, m \geq 0)$.

An import declarations section of a MAD action description is similar to the axioms section in that all of the identifiers used must have been declared prior to this section. On the other hand, an import section *implicitly declares* any identifiers from the module it imports.

In the following sections on sort and constant renaming clauses, we will refer to the module being imported as M_1 and the module importing it as M_2.

D.14.1 Sort Renaming Clauses

A sort renaming clause is of the form

s_1 `is` s_2

where s_1 is a sort which has been declared prior to M_1 and s_2 is a sort declared prior to M_2. Neither of these two sorts may be `Boolean` or an integer range.

(If s_2 were `Boolean` or an integer range, then any objects declared to be of sort s_1 in M_1 would become objects of this built-in sort which has all of its objects predefined.)

A sort may not be renamed more than once in the same import. (i.e. it may not appear on the left hand side of more than one sort renaming clause.)

D.14.2 Constant Renaming Clauses

There are two kinds of constant renaming clauses, depending on whether the constant being renamed is Boolean-valued or not.

Boolean constant renaming clauses

If the constant being renamed is Boolean-valued, a renaming clause is of the form

$c(v_1, ..., v_n)$ `is` *boolean-const-renaming-rhs*$_1$

or

$c(v_1, ..., v_n)$ `is`
`case` $formula_1$ `:` *boolean-const-renaming-rhs*$_1$ `;`
$\vdots$

`case` $formula_k$ `:` *boolean-const-renaming-rhs*$_k$ `;`

[`default :` *boolean-const-renaming-rhs*$_{k+1}$ `;`]

where c is a constant of M_1, $v_1, ..., v_n$ are variables/objects of M_2 or integers, $n \geq 0$, $k \geq 1$. Each variable/object must be of the sort declared for the corresponding argument, or a subsort of that sort, and all variables must occur at most once. Any object appearing as an argument must be "fully instantiated", meaning it cannot have any variable arguments itself. Each $formula_i$ must have no constants and should not contain any variables other than $v_1, ..., v_n$. The part within square brackets is optional. If not included, it is filled in to have *boolean-const-renaming-rhs*$_{k+1}$ be `false`.

In this clause *boolean-const-renaming-rhs*$_i$ is one of

- a Boolean constant term
- *a-Boolean-constant-term* `= true`
- *a-Boolean-constant-term* `= false`
- `true =` *a-Boolean-constant-term*
- `false =` *a-Boolean-constant-term*
- `-`*a-Boolean-constant-term*
- `true`
- `false`

The constant term may not have any other constants appearing as shorthand. (i.e., when viewed as a formula, this term must not be expandable to one in which more than one constant occurs.)

Nonboolean constant renaming clauses

If the constant being renamed is not Boolean-valued, a renaming clause is of the form

$c(v_1, ..., v_n)$ `is` *nonboolean-const-renaming-rhs*$_1$

or

$c(v_1, ..., v_n)$ `is`
`case` $formula_1$ `:` *nonboolean-const-renaming-rhs*$_1$ `;`
$\vdots$
`case` $formula_k$ `:` *nonboolean-const-renaming-rhs*$_k$ `;`
`default :` *nonboolean-const-renaming-rhs*$_{k+1}$ `;`

where c is a constant of M_1, $v_1, ..., v_n$ are variables of M_2, $n \geq 0$, $k \geq 1$. Each variable must be of the sort declared for the corresponding argument (It cannot be a variable of a subsort of the argument declaration), and all variables must occur at most once. Each $formula_i$ must have no constants and should not contain any variables other than $v_1, ..., v_n$.

In this clause *nonboolean-const-renaming-rhs*$_i$ is one of

- a constant term with the same domain as c
- an object which belongs to the domain of c

Additional constraints on constant renaming clauses

There are also some additional constraints depending on the kind of constant appearing on the left hand side of the `is` keyword:

- if it's an action constant, any constant on the right hand side of `is` must be an action constant.
- if it's a statically determined fluent constant, any constant on the right hand side of `is` must be a fluent constant.
- if it's a simple fluent constant, any constant on the right hand side of `is` must be a simple fluent constant.
- if it's a rigid constant, any constant on the right hand side of `is` must be a rigid constant.

The same constant may not be renamed more than once in the same import declaration.

Bibliography

[Akman *et al.*, 2004] Varol Akman, Selim Erdoğan, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Representing the Zoo World and the Traffic World in the language of the Causal Calculator. *Artificial Intelligence*, 153(1–2):105–140, 2004.

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In D. Michie, editor, *Machine Intelligence*, volume 3, pages 131–171. Edinburgh University Press, Edinburgh, 1968.

[Arnold *et al.*, 2000] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language (3rd Edition)*. Addison-Wesley Professional, 2000.

[Balduccini and Gelfond, 2003] Marcello Balduccini and Michael Gelfond. Logic programs with consistency-restoring rules.[4] In *Working Notes of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 2003.

[Balduccini, 2007] Marcello Balduccini. CR-MODELS: An inference engine for CR-prolog. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 18–30, 2007.

[4] `http://www.krlab.cs.ttu.edu/papers/ download/bg03.pdf` .

[Baral and Gelfond, 1997] Chitta Baral and Michael Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31, 1997.

[Baral and Gelfond, 2000] Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domains. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 257–279. Kluwer, 2000.

[Baral *et al.*, 2006] Chitta Baral, Juraj Dzifcak, and Hiro Takahashi. Macros, macro calls, and use of ensembles in modular answer set programming. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 376–390, 2006.

[Baral, 1995] Chitta Baral. Reasoning about actions: Non-deterministic effects, constraints, and qualifications. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2017–2023, 1995.

[Barker *et al.*, 2001] Ken Barker, Bruce Porter, and Peter Clark. A library of generic concepts for composing knowledge bases. In *Proceedings of First International Conference on Knowledge Capture*, pages 14–21, 2001.

[Calimeri *et al.*, 2004] Francesco Calimeri, Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro. A system with template answer set programs. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, 2004.

[Campbell and Lifschitz, 2003] Jonathan Campbell and Vladimir Lifschitz. Reinforcing a claim in commonsense reasoning. In *Working Notes of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 2003.

[Clark and Porter, 1997] Peter Clark and Bruce Porter. Building concept represen-

tations from reusable components. In *Proceedings of AAAI-97*, pages 369–376, 1997.

[Clark and Porter, 2001a] Peter Clark and Bruce Porter. KM — the Knowledge Machine: Situations manual[5] 2001.

[Clark and Porter, 2001b] Peter Clark and Bruce Porter. KM — the Knowledge Machine: User's manual[6] 2001.

[Clark *et al.*, 1996] Peter Clark, Bruce Porter, and Don Batory. A compositional approach to representing planning operators.[7] Technical Report AI06-331, University of Texas at Austin, 1996.

[Doherty and Kvarnström, 2008] Patrick Doherty and Jonas Kvarnström. Temporal action logics. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*. Elsevier, 2008.

[Doğandağ *et al.*, 2004] Semra Doğandağ, Paolo Ferraris, and Vladimir Lifschitz. Almost definite causal theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 74–86, 2004.

[Erdoğan and Lifschitz, 2004] Selim T. Erdoğan and Vladimir Lifschitz. Definitions in answer set programming. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 114–126, 2004.

[Erdoğan and Lifschitz, 2006] Selim T. Erdoğan and Vladimir Lifschitz. Actions

[5] `http://www.cs.utexas.edu/users/mfkb/rkf/km.html` .
[6] `http://www.cs.utexas.edu/users/mfkb/rkf/km.html` .
[7] `http://www.cs.utexas.edu/ftp/pub/AI-Lab/tech-reports/UT-AI-TR-06-331.pdf` .

as special cases. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 377–387, 2006.

[Fellbaum, 1998] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[Fikes and Nilsson, 1971] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[Finger, 1986] Jeffrey Finger. *Exploiting Constraints in Design Synthesis*. PhD thesis, Stanford University, 1986. PhD thesis.

[Geffner, 1990] Hector Geffner. Causal theories for nonmonotonic reasoning. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 524–530. AAAI Press, 1990.

[Gelfond and Lifschitz, 1993] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[Gelfond and Lifschitz, 1998] Michael Gelfond and Vladimir Lifschitz. Action languages.[8] *Electronic Transactions on Artificial Intelligence*, 3:195–210, 1998.

[Gelfond, 2006] Michael Gelfond. Going places — notes on a modular development of knowledge about travel.[9] In *Working Notes of the AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, 2006.

[8] `http://www.ep.liu.se/ea/cis/1998/016/ .`
[9] `http://www.krlab.cs.ttu.edu/papers/ download/gel06.pdf .`

[Giunchiglia and Lifschitz, 1998] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 623–630. AAAI Press, 1998.

[Giunchiglia *et al.*, 2004] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

[Gustafsson and Kvarnström, 2004] Joakim Gustafsson and Jonas Kvarnström. Elaboration tolerance through object-orientation. *Artificial Intelligence*, 153(1–2):239–285, 2004.

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, pages 359–363, 1992.

[Knight and Luk, 1994] Kevin Knight and Steve Luk. Building a large-scale knowledge base for machine translation. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 773–778, 1994.

[Lee and Lifschitz, 2003] Joohyung Lee and Vladimir Lifschitz. Describing additive fluents in action language $\mathcal{C}+$. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1079–1084, 2003.

[Lee and Lifschitz, 2006] Joohyung Lee and Vladimir Lifschitz. A knowledge module: buying and selling. In *Working Notes of the AAAI Symposium on Formalizing Background Knowledge*, 2006.

[Lenat and Guha, 1990] Douglas Lenat and R. V. Guha. *Building large knowledge-based systems.* Addison-Wesley, 1990.

[Lifschitz and Ren, 2006] Vladimir Lifschitz and Wanwan Ren. A modular action description language. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 853–859, 2006.

[Lifschitz *et al.*, 2000] Vladimir Lifschitz, Norman McCain, Emilio Remolina, and Armando Tacchella. Getting to the airport: The oldest planning problem in AI. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 147–165. Kluwer, 2000.

[Lifschitz *et al.*, 2001] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[Lifschitz, 1987] Vladimir Lifschitz. On the semantics of STRIPS. In Michael Georgeff and Amy Lansky, editors, *Reasoning about Actions and Plans*, pages 1–9. Morgan Kaufmann, San Mateo, CA, 1987.

[Lifschitz, 2000] Vladimir Lifschitz. Missionaries and cannibals in the Causal Calculator. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 85–96, 2000.

[Lin and Reiter, 1994] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4:655–678, 1994.

[Lin, 1995] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1985–1991, 1995.

[Matuszek *et al.*, 2006] Cynthia Matuszek, John Cabral, Michael Witbrock, and DeOliveira John. An introduction to the syntax and content of Cyc. In *Working Notes of the AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, 2006.

[McCain and Turner, 1995] Norman McCain and Hudson Turner. A causal theory of ramifications and qualifications. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1978–1984, 1995.

[McCain and Turner, 1997] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 460–465, 1997.

[McCain and Turner, 1998] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 212–223, 1998.

[McCarthy and Hayes, 1969] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[McCarthy, 1959] John McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Reproduced in [McCarthy, 1990].

[McCarthy, 1979] John McCarthy. Ascribing mental qualities to machines. In Mar-

tin Ringle, editor, *Philosophical Perspectives in Artificial Intelligence*. Humanities Press, 1979. Reproduced in [McCarthy, 1990].

[McCarthy, 1987] John McCarthy. Generality in Artificial Intelligence. *Communications of ACM*, 30(12):1030–1035, 1987. Reproduced in [McCarthy, 1990].

[McCarthy, 1990] John McCarthy. *Formalizing Common Sense: Papers by John McCarthy*. Ablex, Norwood, NJ, 1990.

[McCarthy, 1993] John McCarthy. Notes on formalizing context. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 555–560, 1993.

[McCarthy, 2007] John McCarthy. Elaboration tolerance.[10] In progress, 2007.

[Minsky, 1975] Marvin Minsky. A framework for representing knowledge. In Patrick Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, New York, 1975.

[Nau *et al.*, 1999] Dana Nau, Yue Cao, Amnon Lotem, and Héctor Munõz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 968–973, 1999.

[Parmar, 2001] Aarati Parmar. The representation of actions in KM and Cyc. Technical Report FRG-1, Department of Computer Science, Stanford University, 2001.

[Pednault, 1987] Edwin Pednault. Formulating multi-agent, dynamic world problems in the classical planning framework. In Michael Georgeff and Amy Lansky, editors, *Reasoning about Actions and Plans*, pages 47–82. Morgan Kaufmann, San Mateo, CA, 1987.

[10] `http://www-formal.stanford.edu/jmc/elaboration.html` .

[Pednault, 1988] Edwin Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, 1988.

[Pednault, 1994] Edwin Pednault. ADL and the state-transition model of action. *Journal of Logic and Computation*, 4:467–512, 1994.

[Ramachandran *et al.*, 2005] Deepak Ramachandran, Pace Reagan, and Keith Goolsbey. First-orderized ResearchCyc: expressivity and efficiency in a common-sense ontology. In *Papers from the AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications.*, 2005.

[Robinson, 1965] Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12:23–41, 1965.

[Sergot and Craven, 2005] Marek Sergot and Robert Craven. Some logical properties of nonmonotonic causal theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 198–210, 2005.

[Shanahan, 1997] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[Shepard *et al.*, 2005] Blake Shepard, Cynthia Matuszek, C. Bruce Fraser, William Wechtenhiser, David Crabbe, Zelal Güngördü, John Jantos, Todd Hughes, Larry Lefkowitz, Michael Witbrock, Doug Lenat, and Erik Larson. A Knowledge-Based approach to network security: Applying Cyc in the domain of network risk assessment. In *Proceedings of the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 1563–1568, 2005.

[Stroustrup, 2000] Bjarne Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley Professional, 2000.

[Thielscher, 1997] Michael Thielscher. Ramification and causality. *Artificial Intelligence*, 89(1–2):317–364, 1997.

[Turner, 1997] Hudson Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

[Turner, 2004] Hudson Turner. Strong equivalence for causal theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 289–301, 2004.

[Veloso, 1992] Manuela Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Carnegie Mellon University, 1992. PhD thesis.

Vita

Selim Turhan Erdoğan was born in Ankara, Turkey, in 1977, and lived in Ankara until moving to Austin in 2000. He attended T.E.D. Ankara Koleji up to the end of high school. He received the B.S. degree in Electrical and Electronic Engineering from Middle East Technical University and the M.S. degree in Computer Engineering and Information Science from Bilkent University. From 2000 onwards, he has been enrolled in the doctoral program in Computer Sciences at the University of Texas at Austin.

Permanent Address: Bilkent University Lojman 8/5
06800 Bilkent, Ankara
TURKEY

This dissertation was typeset with LaTeX 2_ε[11] by the author.

[11] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.