The Dissertation Committee for Wanwan Ren

certifies that this is the approved version of the following dissertation:

A Modular Language for Describing Actions

Committee:

Vladimir Lifschitz, Supervisor

Robert Boyer

Michael Gelfond

Gordon Novak

Bruce Porter

A Modular Language for Describing Actions

by

Wanwan Ren, 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

December 2009

To My Family

Acknowledgements

I am sincerely grateful to my advisor Vladimir Lifschitz. His insightful guidance and continuous encouragement were with me throughout my research work. His observations often inspired me. His help was something I could always count on, from big pictures of my work, to a detailed step in proving a proposition. This dissertation would not be possible without him.

I would like to thank all the other members of my dissertation committee, Robert Boyer, Michael Gelfond, Gordon Novak and Bruce Porter for serving on my committee and for their invaluable comments and suggestions.

I wish to thank all my colleagues Esra Erdem, Selim Erdoğan, Paolo Ferraris, Joohyung Lee, Yulia Lierler, and Fangkai Yang. It was a fortune for me to have them in the same research group. Conversations with them benefited me both in research and in life. Particularly, I am grateful to Selim for his useful comments on this dissertation. Special thanks also go to Selim, Yulia and Fangkai for their help with lodging when I traveled to Austin for the last parts of my dissertation work.

My thanks extend to my family members for their encouragement and support. My parents always keep my spirits up and energize me whether they are with me or from thousands of miles away. They also helped with childcare and housework. My wife Mei often stimulated me to keep me working highly motivated. Especially at the late stage of wrapping up my dissertation, she took every effort to let me concentrate on my work. My brother Jianhua often encouraged me and

offered me suggestions regarding doctoral study based on his own experience. Last but not least, my thanks go to my beloved son Kevin for the endless joy he brings to me.

This research has been partially supported by The National Science Foundation.

WANWAN REN

The University of Texas at Austin

December 2009

A Modular Language for Describing Actions

Publication No. _____

Wanwan Ren, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Vladimir Lifschitz

This dissertation is about the design of a modular language for describing actions. The modular action description language, MAD, is based on the action language $\mathcal{C}+$. In this new language, the possibility of "importing" a module allows us to describe actions by referring to descriptions of related actions introduced earlier, rather than by listing all effects and preconditions of every action explicitly. The use of modular action descriptions eliminates the need to reinvent theories of similar domains over and over again. Another advantage of this representation style is that it is similar to the way humans describe actions in terms of other actions.

We first define the syntax of a fragment of MAD, called mini-MAD, and then extend it to the full version of MAD. The semantics of mini-MAD is defined by grounding action descriptions and translating them into $\mathcal{C}+$. However, for the full

version of MAD, it would be difficult to define grounding. Instead, we use a new approach to the semantics of variables in action descriptions, which is based on more complex logical machinery—first-order causal logic. Grounding is important as an implementation method, but we argue that it should be best avoided in the definition of the semantics of expressive action languages. We show that, in application to mini-MAD, the two semantics are equivalent.

Furthermore, we prove that MAD action descriptions have some desirable, intuitively expected mathematical properties.

We hope that MAD will make it possible to create a useful general-purpose library of standard action descriptions and will contribute in this way to solving the problem of generality in Artificial Intelligence.

Contents

Chapter 1

Introduction

Knowledge about actions is an important part of commonsense knowledge in the area of Artificial Intelligence. For decades, researchers tried to describe how actions affect the states of world and to correctly and efficiently reason about actions. In recent years, significant progress was made in the study of actions. In particular, the frame problem [McCarthy and Hayes, 1969] has been solved using nonmonotonic knowledge representation formalisms [Shanahan, 1997]. This theoretical work has led to the creation of several implemented systems with very expressive input languages that can be used to solve computational problems related to actions, such as prediction, postdiction and planning. Theories of causality by Geffner [1990], Lin [1995], McCain and Turner [1997] allow us to express causal dependencies between fluents, which is essential for solving the ramification problem [Finger, 1986].

Action description languages play a key role in the study of actions. These languages allow us to represent knowledge about actions more concisely than other formalisms, such as the situation calculus [McCarthy and Hayes, 1969]. Many action description languages have been described in the literature, from the well-known

STRIPS [Fikes and Nilsson, 1971], to more expressive ADL [Pednault, 1994], to action languages invented in recent years, such as $\mathcal{C}+$ [Giunchiglia *et al.*, 2004].

Unfortunately, this work has not yet contributed to solving another important problem—the problem of generality in AI [McCarthy, 1987]. As observed in [Erdoğan and Lifschitz, 2005], it is common for humans to describe an action by referring to another more "general" action. For instance, the dictionary explains "walk" as "move by foot", and "climb" as "go up or down". These explanations of the words "walk" and "climb" do not characterize these actions in terms of their effects, instead, the actions are presented as special cases of some other actions that are supposed to be already familiar to us. The most fundamental actions still need to be described directly in terms of the changes that they cause. The action "move", for instance, means to "cause to change position" according to the dictionary. But in most cases the natural way to describe an action is to relate it to more basic actions. The action languages designed in the past do not allow us to do this.

Describing every action directly in terms of its effects is similar to using a programming language without procedures or functions. When subroutines are not available, the programmer is not able to "factor out" common parts of similar programs. In the same way, the action languages designed in the past force researchers to repeatedly reinvent theories of similar physical domains.

Therefore, it is important to make descriptions of actions reusable. This idea has led to the design of the *M*odular *A*ction *D*escription (MAD) language in this dissertation. This new language MAD allows us to describe actions by referring to related action descriptions introduced earlier. MAD is modular; each module describes an action or a group of related actions. For a "basic" action, the corresponding module lists its effects and preconditions, as in the action languages

that were used in the past. But when we want to describe an action by referring to other related actions, MAD provides the possibility of referring to other modules by "importing" them. The import statement is the main new syntactic feature of MAD. A description of an action domain in MAD is generally a list of modules referring to each other, and some of them may belong to a general-purpose library.

We design the action description language MAD on the basis of the action description language $\mathcal{C}+$ [Giunchiglia *et al.*, 2004]. We first present a fragment of MAD, called mini-MAD. The syntax of mini-MAD stresses the modular structure of an action description, but it doesn't contain some of the more complex constructs. The semantics of mini-MAD is defined in two steps. First we define how to turn any action description into a single module by replacing each import statement with a modified form of the module that is being imported. Second, we define how to turn any single-module mini-MAD action description into a $\mathcal{C}+$ action description by grounding. These two steps together translate mini-MAD into $\mathcal{C}+$. This approach was presented in the first publication about MAD [Lifschitz and Ren, 2006].

Then we present the full version of MAD, which adds to mini-MAD several new syntactical constructs. The semantics of MAD is defined in two steps also, but the second step uses a totally different approach [Lifschitz and Ren, 2007], as grounding is not feasible for full MAD with its richer syntax. This new approach uses a translation into first-order causal logic [Lifschitz, 1997].

In [Lifschitz and Ren, 2006], we predicted that it would be possible to use MAD to build a library of reusable standard descriptions for "basic" actions. This was confirmed recently by Erdoğan [2008], who built a general-purpose library for action descriptions in a dialect of MAD and developed a system for automated reasoning about MAD action descriptions, which was used to test the library.

In this dissertation, we first review the problem of generality in Artificial Intelligence, difficulties in reasoning about actions, the evolution and characteristics of action languages, and research on representing actions by logic programs in Chapter 2. Chapter 3 reviews propositional causal logic defined in [Giunchiglia *et al.*, 2004]. Chapter 4 reviews the action language $\mathcal{C}+$ and its implementation CCALC.

We begin to present our original work in Sections 5.1 and 5.2 and Chapter 6. First we illustrate, by an example, the main features of MAD—describing actions by modules and the capability of referring to other actions by importing modules; then we define the syntax of mini-MAD. Chapter 7 presents the two-step semantics of mini-MAD: eliminating import statements followed by grounding.

In Chapter 8 we define the syntax of full MAD. Chapter 10 defines the semantics of MAD by representing action descriptions as causal theories in the sense of first-order causal logic [Lifschitz, 1997], which is reviewed in Chapter 9.

In Chapter 11, we state and prove three propositions confirming that the semantics of MAD has some natural, intuitively expected properties. Chapter 12 shows that the semantics of MAD based on first-order causal logic, when restricted to mini-MAD, is equivalent to the semantics from Chapter 7 that uses grounding.

Chapter 13 reviews related work on adding modularity to existing formalisms for describing actions and recent work on the use of MAD. Chapter 14 concludes this dissertation and presents some directions for future work.

Chapter 2

Background

2.1 Generality in Artificial Intelligence

The problem of "generality in AI" was described by John McCarthy in his Turing Award lecture [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.

Work on large-scale knowledge-based systems [Lenat and Guha, 1990, Knight and Luk, 1994, Fellbaum, 1998, Barker *et al.*, 2001] deals, among other things, with encoding general-purpose knowledge and thus contributes to solving the problem of generality in AI. But this work has been almost completely disjoint from work on designing expressive action languages. Facts about moving objects around and facts about buying and selling, mentioned by McCarthy in the quote above, belong to the area of knowledge about actions. Solving the problem of generality with focus on actions and their effects on fluents—features of the world that may change with time—is the main motivation of this dissertation.

2.2 Problems in Reasoning about Actions

Reasoning about actions is an important sub-area of commonsense reasoning. It involves describing how actions affect the states of the world. Computer systems are supposed to derive relevant conclusions from properly axiomatized commonsense knowledge about actions [McCarthy, 1959].

AI researchers often use "toy worlds" to illustrate and test their ideas related to describing actions and changes before applying them to large-scale domains. The

following domains are among the toy worlds used in this dissertation.

- The Blocks World. In this domain several blocks may be located on the table or on the top of each other. Available actions are moving a block and sometimes painting it to a different color. Performing these actions changes the location of the blocks, the status of which block is on the top of which one, or the color of the blocks.

- Monkey and Bananas. A monkey wants to get a bunch of bananas hanging from the ceiling beyond his reach. There is a box available in the room. He can get the bananas by first walking to the box, then pushing the box under the bananas, then climbing on the top of the box, and finally grasping the bananas.

- The Robot. A robot can walk or carry an object to a different location.

- The Suitcase [Lin, 1995]. A suitcase has locks, which can be at one of two positions. The action toggle can be executed to change the position of a lock. If all locks are up, the suitcase is opened.

Serious difficulties were recognized during attempts to describe such action domains using formal logic. One of these problems is the frame problem [McCarthy and Hayes, 1969]—describing what remains unchanged after executing an action. For instance, in the Blocks World domain, if one block is moved, its location changes but its color doesn't, and every other block remains at the same location. In large domains, there are usually many fluents that do not change when an action is performed, so it would be neither feasible nor commonsensical to enumerate all of them.

The frame problem can be solved by formalizing what McCarthy called "the commonsense law of inertia": by default, properties of objects are assumed to remain unchanged with passage of time. Defaults are nonmonotonic: when a new postulate is added, conclusions derived from a default may need to be retracted. This makes the logic of defaults different from classical logic, in which, when new axioms are added we can only derive more, but never less. In particular, the commonsense law of inertia is nonmonotonic. For instance, the conclusion that moving a block to a new location does not affect its color will have to be retracted if we learn that this location is a bucket with paint. Several early papers on formal nonmonotonic reasoning were published in [Artificial Intelligence, 1980].

Another challenge, called the ramification problem [Finger, 1986], is to describe indirect effects of an action. For example, if a robot walks to another location while holding a block, then the location of the block changes also. Even though no action directly affects the block's location, the change of the robot's location causes the block to move also, given that it is held by the robot. Another example: when a block is immersed in paint, its color is that of the paint; when the action of putting the block in a bucket with paint is performed, the location of the block changes as the direct effect of the action, and its color changes as the indirect effect (if its original color was different). The ramification problem is related to the frame problem—the latter becomes more difficult when actions with indirect effects are present.

The ramification problem has been solved by nonmonotonic theories of causality [Geffner, 1990, Lin, 1995, McCain and Turner, 1997, Giunchiglia *et al.*, 2004]. The causal logic defined in the last of these papers is used in the definition of $\mathcal{C}+$, and it is reviewed in Chapter 3.

2.3 Action Description Languages

As defined by Gelfond and Lifschitz [1998], action description languages are formal models of parts of natural languages that are used for talking about the effects of actions. They usually have easily understandable and concise syntax. Central to this idea is the concept of a transition system. A transition system can be thought of as a labeled directed graph. Every vertex corresponds to a state; every edge is labeled by actions that lead from the state represented by the starting vertex to the state represented by the ending vertex. Semantically, action description languages define transitions systems.

2.3.1 STRIPS and ADL

The STRIPS language [Fikes and Nilsson, 1971] is not an action language in the sense of [Gelfond and Lifschitz, 1998] because STRIPS operators operate on "world models" instead of states of a transition system, but it is closely related. In STRIPS an action is described by its preconditions, its add list and its delete list. For instance, the STRIPS description of the action $Walk(x, y)$ (the robot walks from x to y) in the Robot domain may look like this:

Preconditions: $At(x)$, $x \neq y$,

Add list: $At(y)$,

Delete list: $At(x)$.

STRIPS provides a built-in solution to the frame problem—when an action is executed, any fluent not included in its add list or its delete list remains unchanged. The STRIPS system uses a resolution theorem prover to perform planning—to find a sequence of operators that transforms the initial world model into a model that satisfies a given goal formula.

The expressivity of STRIPS is limited. Pednault [1987] observed that the expressive power of STRIPS can be enhanced by allowing the effects of an operator to be "conditional". For instance, in the suitcase domain when the toggle action is performed on a lock, the position of that lock changes from up to down or the other way around. The effect of this toggle action is conditional: it depends on the position of the lock when the action is executed. We are unable to describe it in STRIPS. The ADL action schema for describing $Toggle(y)$ may look like this:

Precondition $Precond^{Toggle}(y)$: $True$,
Add condition $Add_{Up}^{Toggle}(x, y)$: $(x = y) \wedge \neg Up(x)$,
Delete condition $Delete_{Up}^{Toggle}(x, y)$: $(x = y) \wedge Up(x)$.

Thus $Toggle(y)$ is always executable; $Up(x)$ becomes true after executing this action if $(x = y) \wedge \neg Up(x)$ is true; $Up(x)$ becomes false after executing this action if $(x = y) \wedge Up(x)$ is true.

Moreover, ADL is more expressive than STRIPS in that it allows non-Boolean fluents, while in STRIPS every fluent must be Boolean. For example, we can describe the suitcase domain using a non-Boolean fluent $Position$. Now the universe consists of objects of two kinds: locks and levels; the symbols Up and $Down$ now become object constants representing levels. Here is the corresponding version of the ADL action schema for $Toggle(y)$:

Precondition $Precond^{Toggle}(y)$: $Lock(y)$,

Update condition $Update^{Toggle}_{Position}(x, y, z)$:

$$Lock(x) \wedge Level(z) \wedge (x = y) \wedge ((Position(x) = Up \wedge z = Down) \\ \vee (Position(x) = Down \wedge z = Up)).$$

2.3.2 $\mathcal{A}$, $\mathcal{C}$ and $\mathcal{C}+$

In action language $\mathcal{A}$ [Gelfond and Lifschitz, 1993], an action description is a set of *propositions*. Each proposition describes the effect of one action on one fluent. This is more concise and more intuitive than the syntax of STRIPS and ADL. For instance, the $\mathcal{A}$ propositions for describing the action $Toggle(x)$ in the suitcase domain are:

$$\begin{aligned} &Toggle(x) \textbf{ caused } Up(x) \textbf{ if } \neg Up(x), \\ &Toggle(x) \textbf{ caused } \neg Up(x) \textbf{ if } Up(x). \end{aligned} \tag{2.1}$$

Language $\mathcal{A}$ is propositional; variables like x in this example are schematic variables used to describe finite sets of propositions that follow the same pattern. $\mathcal{A}$ is as expressive as the propositional fragment of ADL.

That work, along with the theory of nonmonotonic causal reasoning presented in [McCain and Turner, 1997], has led to the design of action language $\mathcal{C}$ [Giunchiglia and Lifschitz, 1998] and its extension $\mathcal{C}+$ [Giunchiglia *et al.*, 2004]. Language $\mathcal{C}$ is essentially a superset of language $\mathcal{A}$, and it is more expressive. In particular, the use of static causal laws in $\mathcal{C}$ provides a way to describe indirect effects of actions, which solves the ramification problem. For example, if a robot holds a block, then there is a causal dependency between the location of the block and the location of the robot:

$$\textbf{caused } At(Block, l) \textbf{ if } At(Robot, l) \wedge Holds(Robot, Block). \tag{2.2}$$

Consider an action, such as walking, that affects the location of the robot:

$$Walk(l) \textbf{ causes } At(Robot, l). \tag{2.3}$$

Static causal law (2.2), in combination with the dynamic causal law (2.3), implies that a change in the location of the block may be an indirect effect of walking.

Language $\mathcal{C}+$ [Giunchiglia *et al.*, 2004] extended $\mathcal{C}$ by adding symbols for non-Boolean fluents. For instance, the ADL update condition shown at the end of Section 2.3.1 can be represented in $\mathcal{C}+$ as follows:

$$\begin{aligned} &Toggle(x) \textbf{ causes } Position(x) = Up \textbf{ if } Position(x) = Down, \\ &Toggle(x) \textbf{ causes } Position(x) = Down \textbf{ if } Position(x) = Up. \end{aligned} \tag{2.4}$$

$\mathcal{C}+$ is also more expressive than $\mathcal{C}$ in some other ways. In $\mathcal{C}+$ we can describe the effects that an action has in all states that are not "abnormal". For instance, the $\mathcal{C}+$ causal law

$$Toggle(x) \textbf{ causes } Position(x) = Up \textbf{ if } Position(x) = Down \textbf{ unless } Ab(x)$$

expresses that toggling a lock normally changes its position from *Down* to *Up*, but exceptions are possible. If no information about abnormal states is included in the action description then this "defeasible" causal law has the same effect as the first of the causal laws (2.4). On the other hand, we can express that a state is abnormal (with respect to the lock x) whenever the lock x is broken by writing

$$\textbf{caused } Ab(x) \textbf{ if } Broken(x).$$

The Causal Calculator (CCALC) is a software system that solves planning and automated reasoning problems for action domains described in $\mathcal{C}+$ [McCain, 1997, Lee, 2005]. $\mathcal{C}+$ and CCALC are discussed in more detail in the next chapter.

2.4 Logic Programs and Reasoning about Actions

Logic programming became a member of the family of nonmonotonic reasoning systems after the semantics of "negation as failure" was clarified. Negation as failure is a nonmonotonic operator. For instance, the atom p is the consequence of the one-rule logic program

$$p \leftarrow \text{not } q.$$

If the fact q is added as the second rule, then conclusion p will be retracted.

Among the available definitions of the meaning of negation as failure, influential are the completion semantics [Clark, 1978], the well-founded semantics [Van Gelder *et al.*, 1991], and the answer set (stable model) semantics [Gelfond and Lifschitz, 1988, Gelfond and Lifschitz, 1991]. Action language $\mathcal{A}$ was shown to be related to logic programming under the answer set semantics [Gelfond and Lifschitz, 1993]. For instance, $\mathcal{A}$ propositions (2.1) can be translated into the following logic program:

$$\begin{array}{rcl} Up(x,t+1) & \leftarrow & Toggle(x,t),\ \neg Up(x,t), \\ \neg Up(x,t+1) & \leftarrow & Toggle(x,t),\ Up(x,t). \end{array} \tag{2.5}$$

Similar translations are available for a language that permits the concurrent execution of actions [Baral and Gelfond, 1997], and for a language with static causal laws [Turner, 1997]. The answer set semantics is also closely related to the system of causal logic proposed in [McCain and Turner, 1997], mentioned above in connection

with action language $\mathcal{C}$ [McCain, 1997, Section 6.3.2].

There are many implemented systems for computing answer sets available, such as SMODELS [Niemelä and Simons, 2000] and DLV [Eiter *et al.*, 1998]. Such systems are called answer set solvers, and their use for solving search problems is known as answer set programming [Lifschitz, 1999, Marek and Truszczyński, 1999, Niemelä, 1999]. Therefore a translation from an action language to answer set programming gives us the possibility to reason about actions using answer set solvers. A translation from causal theories of a special kind, called definite, into the language of answer set programming is available [McCain, 1997]. This translation has been extended to be able to translate from causal theories of a more general kind, called almost definite [Doğandağ *et al.*, 2004].

Chapter 3

Review of Causal Logic

In this chapter we review the concept of a causal theory [Giunchiglia *et al.*, 2004, Section 2], which is used in the definition of the semantics of $\mathcal{C}+$ in the next chapter.

3.1 Causal Theories

A *(multi-valued propositional) signature* is a set σ of symbols called *constants*, along with a nonempty finite set $Dom(c)$ of symbols, disjoint from σ, assigned to each constant c. We call $Dom(c)$ the *domain* of c. A constant is *Boolean* if its domain is $\{\mathbf{false}, \mathbf{true}\}$. An *atom* of a signature σ is an expression of the form $c{=}v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. For a Boolean constant p, atoms $p = \mathbf{false}$ and $p = \mathbf{true}$ can be abbreviated as $\neg p$ and p respectively. A *formula* of σ is a propositional combination of atoms.

Begin with a multi-valued propositional signature σ. A (*causal*) *rule* is an expression of the form $F \Leftarrow G$ ("F is caused if G is true"), where F and G are formulas of σ, called the *head* and the *body* of the rule. Rules with the head $\bot$ (falsity) are called *constraints*. Syntactically, a (multi-valued propositional) *causal*

theory is a set of causal rules.

Informally, the semantics of causal theories is based on the idea that "anything is true if and only if it is caused"; the "only if" part expresses the "Principle of Universal Causation" [McCain and Turner, 1997]. Mathematically the semantics is defined as follows.

An *interpretation* of σ is a function that maps every element c of σ to an element of $Dom(c)$. An interpretation I *satisfies* an atom $c=v$ (symbolically, $I \models c=v$) if $I[c] = v$. We will sometimes identify an interpretation with the set of atoms that are satisfied by it. The satisfaction relation is extended from atoms to arbitrary formulas according to the usual truth tables for the propositional connectives.

Let T be a causal theory, and let I be an interpretation of its signature. The *reduct* T^I of T relative to I is the set of the heads of all rules in T whose bodies are satisfied by I. We say that I is a *model* of T if I is the unique model of T^I. We say that a formula F is *entailed* by a causal theory T if every model of T satisfies F. This concludes the definition of the semantics of causal theories.

Intuitively, the reduct T^I is the set of formulas that are caused under interpretation I according to the rules of T. Because anything that is caused must be true, I must satisfy T^I. Because of the Principle of Universal Causation, every formula that is satisfied by I must be entailed by T^I, which means that an interpretation different from I cannot satisfy T^I.

3.2 Examples

Take $\sigma_1 = \{p, q\}$ where p and q are Boolean constants, and let the causal rules of T_1 be

$$\begin{aligned} p &\Leftarrow q, \\ q &\Leftarrow q, \\ \neg q &\Leftarrow \neg q. \end{aligned} \tag{3.1}$$

Consider the interpretation I defined by

$$I[p] = I[q] = \mathbf{true}. \tag{3.2}$$

Then

$$T_1^I = \{p, q\},$$

so I is the unique model of T_1^I, and thus a model of T_1.

If we take I to be any of the other three interpretations of the signature σ_1, we will see that either I doesn't satisfy T_1^I, or T_1^I is satisfied by more than one interpretations. Therefore interpretation (3.2) is the only model of T_1.

Informally, causal rules (3.1) assert that, under various conditions, there is a cause for p, q, or $\neg q$. But no rule in (3.1) may lead to the conclusion that there is a cause for $\neg p$. Consequently,

$\neg p$ is not caused.

By the principle of universal causation, we can conclude that

$\neg p$ is not true

or, equivalently,

$$p \text{ is true.}$$

Again by the principle of universal causation,

$$p \text{ is caused.}$$

The only way to use rules (3.1) to establish that p is caused is to refer to first rule

$$p \Leftarrow q,$$

which says: p is caused if q is true. Consequently,

$$q \text{ is true.}$$

Therefore the interpretation making both p and q true is the only possible model of T_1. And it is indeed a model, because, under this interpretation, the first two rules of (3.1) guarantee the existence of causes both for p and for q.

Consider now an example with non-Boolean constants. Let

$$\sigma_2 = \{c\},\ Dom(c) = \{1, \ldots, n\}$$

for some positive integer n, and let the only rule of T_2 be

$$c{=}1 \Leftarrow c{=}1. \tag{3.3}$$

The interpretation I defined by $I[c] = 1$ is a model of T_2. Indeed,

$$T_2^I = \{c{=}1\},$$

so that I is the only model of T_2^I. Furthermore, T_2 has no other models. Indeed, for any interpretation J such that $J[c] \neq 1$, T_2^J is empty, and I is a model of T_2^J different from J.

It follows that causal theory T_2 entails $c{=}1$.

Consider now what happens if we add the rule

$$c{=}2 \Leftarrow \top \tag{3.4}$$

to this theory. Denote the extended theory by T_3. The reduct of T_3 relative to any interpretation includes the atom $c{=}2$. Consequently, the interpretation assigning 2 to c is the only possible model of T_3. It is easy to see that this is indeed a model.

The causal theory T_3 does not entail $c = 1$; it entails $c = 2$. This example shows that the logic introduced above is nonmonotonic.

Intuitively, rule (3.3) expresses that 1 is "the default value" of c, and rule (3.4) overrides this default.

3.3 Exogenous Constants

In this section we show how causal logic can be used to express an idea that plays an important role in formalizing properties of actions and fluents.

Consider the causal theory T_4 whose signature is σ_2 and whose causal rules

are

$$\begin{aligned} c = 1 &\Leftarrow c = 1, \\ &\vdots \\ c = n &\Leftarrow c = n. \end{aligned} \tag{3.5}$$

It is easy to check that any of the n possible interpretations of σ_2 is a model of T_4. Intuitively, these causal rules express that if c has any value $v \in \{1, \ldots, n\}$, there is a cause for that. T_4 disables the principle of universal causation for c; it makes c "exogenous".

When action domains are represented in causal logic, rules of this kind are often used to express that the values of fluents at time 0 are exogenous. (Values of fluents at other time instants are not treated as exogenous: they are caused either by the execution of actions or by inertia.) They are also often used to express that the execution of actions is exogenous: whether or not an action is executed, there is a cause for this.

3.4 Inertia

The following example describes a domain that involves a fluent c with values $\{1, \ldots, n\}$, two time instants 0 and 1, and an action a that may be executed between the two time instants. Causal theory T_5 has the signature

$$\sigma_3 = \{c_0, c_1, a\},\ Dom(c_0) = Dom(c_1) = \{1, \ldots, n\},\ Dom(a) = \{\mathbf{false}, \mathbf{true}\}$$

for some positive integer n, and its causal rules are

$$\begin{aligned}
c_0 = 1 &\Leftarrow c_0 = 1,\\
&\vdots\\
c_0 = n &\Leftarrow c_0 = n,\\
a &\Leftarrow a,\\
\neg a &\Leftarrow \neg a,\\
c_1 = n &\Leftarrow a,\\
c_1 = 1 &\Leftarrow c_0 = 1 \wedge c_1 = 1,\\
&\vdots\\
c_1 = n &\Leftarrow c_0 = n \wedge c_1 = n.
\end{aligned} \tag{3.6}$$

The first $n + 2$ rules express that the initial value of c and the execution of a are exogenous. The next rule says that the execution of a causes c to take the value n. The last n rules express the commonsense law of inertia (Section 2.2): if the value of c at time 1 is the same as the value of c at time 0, then there is a cause for c to have that value at time 1. Intuitively inertia is the cause.

T_5 has $2n$ models. In n of them a is not executed and the value of c at time 1 is the same as at time 0:

$$I[c_0] = I[c_1] = k,\ I[a] = \mathbf{f} \qquad (k \in \{1, \ldots, n\}).$$

In the other n models, a is executed and the value of c at time 1 equals n:

$$I[c_0] = k,\ I[c_1] = n,\ I[a] = \mathbf{true} \qquad (k \in \{1, \ldots, n\}).$$

Chapter 4

Review of Action Language $\mathcal{C}+$ and its Implementation CCalc

According to the semantics of $\mathcal{C}+$, every action description represents a transition system. The semantics is defined by translating action descriptions into nonmonotonic causal logic discussed above. In this chapter we review the syntax and semantics of $\mathcal{C}+$ [Giunchiglia *et al.*, 2004, Section 4] and the implementation of $\mathcal{C}+$, called Causal Calculator (CCalc).[1]

4.1 Fluents and Actions in $\mathcal{C}+$

The *signature* of an action description in $\mathcal{C}+$ is a signature in the sense of Section 3.1, with its constants divided into two groups—the set σ^{fl} of *fluent constants* and the set σ^{act} of *action constants*. Fluent constants are further partitioned into *simple* and *statically determined*.

The need to distinguish between simple fluents and statically determined

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

fluents is due to the fact that some fluents are directly affected by actions, and some fluents are characterized in terms of other fluents. For example, in the Blocks World domain, the fluent $On(x, y)$ (block x is on top of block y) is a simple fluent; it can be directly affected by moving block x. The statically determined fluent $Clear(x)$ (there are no blocks on top of block x) is defined in terms of On, and it can be only affected by actions indirectly.

For simplicity we consider here the case when all action constants are Boolean. Intuitively, all actions to which the value **true** is assigned are understood to be executed concurrently.

4.2 Syntax and Semantics of $\mathcal{C}+$

In $\mathcal{C}+$ two special kinds of formulas are distinguished: A *fluent formula* is a formula such that all constants occurring in it are fluent constants; an *action formula* is a formula that contains at least one action constant and no fluent constants.

According to the syntax of the action language $\mathcal{C}+$ [Giunchiglia *et al.*, 2004, Section 4], an *action description* is a set of "causal laws" that are constructed from formulas and keywords. In $\mathcal{C}+$ there are causal laws of three kinds—*static* laws, *action dynamic* laws, and *fluent dynamic* laws. A *static law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \tag{4.1}$$

where F and G are fluent formulas. An *action dynamic law* is an expression of the form (4.1) in which F is an action formula and G is a formula. A *fluent dynamic law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{4.2}$$

where F is a fluent formula not containing statically determined constants, G is a fluent formula, and H is a formula.

For any action description D and any nonnegative integer m, we will define the causal theory D_m in the sense of Section 3.1. Intuitively, each model of D_m corresponds to a path of length m in the transition system represented by D.

The signature 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 number i is the "time stamp" of the constant $i : c$.) The domain of $i\!:\!c$ is the same as the domain of c. By $i\!:\!F$ we denote the result of inserting $i\!:$ in front of every occurrence of every constant in a formula F, and similarly for a set of formulas. The rules of D_m are:

$$i\!:\!F \Leftarrow i\!:\!G \tag{4.3}$$

for every static law (4.1) in D and every $i \in \{0, \ldots, m\}$, and for every action dynamic law (4.1) in D and every $i \in \{0, \ldots, m-1\}$;

$$i{+}1\!:\!F \Leftarrow (i{+}1\!:\!G) \wedge (i\!:\!H) \tag{4.4}$$

for every fluent dynamic law (4.2) in D and every $i \in \{0, \ldots, m-1\}$;

$$0\!:\!c{=}v \Leftarrow 0\!:\!c{=}v \tag{4.5}$$

for every simple fluent constant c and every $v \in Dom(c)$.

Note that the definition of D_m treats simple fluent constants and statically determined fluent constants in different ways: rules (4.5), expressing that c is ex-

ogenous at time 0, are included only when c is simple.

A *state* is an interpretation s of σ^{fl} such that $0\!:\!s$ is a model of D_0. States are the vertices of the transition system represented by D. A *transition* is a triple $\langle s, e, s' \rangle$, where s and s' are interpretations of σ^{fl} and e is an interpretation of σ^{act}, such that $0\!:\!s \cup 0\!:\!e \cup 1\!:\!s'$ is a model of D_1. Transitions correspond to the edges of the transition system: for every transition $\langle s, e, s' \rangle$, it contains an edge from s to s' labeled e. These labels e will be called *events*.

4.3 Example

In the example below, we will use some abbreviations for causal laws. An expression of the form

$$F \textbf{ causes } G$$

stands for the fluent dynamic law

$$\textbf{caused } G \textbf{ if } \top \textbf{ after } F.$$

An expression of the form

$$\textbf{exogenous } a,$$

where a is an action constant, stands for the pair of action dynamic laws

$$\textbf{caused } a \textbf{ if } a,$$

$$\textbf{caused } \neg a \textbf{ if } \neg a.$$

An expression of the form

$$\textbf{inertial } c,$$

where c is a simple fluent constant, stands for the set of fluent dynamic laws

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

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

Now consider the following $\mathcal{C}+$ action description SD:

a **causes** p,

exogenous a,

inertial p,

where a is an action constant and p is a Boolean simple fluent constant. Written in full, the causal laws in SD are

caused p **if** $\top$ **after** a,

caused a **if** a,

caused $\neg a$ **if** $\neg a$,

caused p **if** p **after** p,

caused $\neg p$ **if** $\neg p$ **after** $\neg p$.

The causal rules of the corresponding causal theory SD_0 are

$$0\!:\!p \Leftarrow 0\!:\!p,$$
$$0\!:\!\neg p \Leftarrow 0\!:\!\neg p.$$

This causal theory has two models. In one of them $0 : p$ is false and in the other it is true.

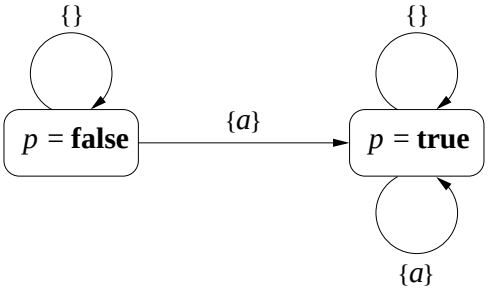

Figure 4.1: The transition system corresponding to the action description *SD*.

The causal rules of the corresponding causal theory SD_1 are

$$1{:}p \Leftarrow 0{:}a,$$
$$0{:}p \Leftarrow 0{:}p,$$
$$0{:}\neg p \Leftarrow 0{:}\neg p,$$
$$0{:}a \Leftarrow 0{:}a,$$
$$0{:}\neg a \Leftarrow 0{:}\neg a,$$
$$1{:}p \Leftarrow (0{:}p) \wedge (1{:}p),$$
$$1{:}\neg p \Leftarrow (0{:}\neg p) \wedge (1{:}\neg p).$$

This causal theory has 4 models:

$$0{:}p = \textbf{false},\ 0{:}a = \textbf{false},\ 1{:}p = \textbf{false},$$
$$0{:}p = \textbf{true},\ 0{:}a = \textbf{false},\ 1{:}p = \textbf{true},$$
$$0{:}p = \textbf{false},\ 0{:}a = \textbf{true},\ 1{:}p = \textbf{true},$$
$$0{:}p = \textbf{true},\ 0{:}a = \textbf{true},\ 1{:}p = \textbf{true}.$$

Accordingly, action description *SD* represents the transition system shown in Figure 4.1 [Giunchiglia *et al.*, 2004, Figure 1]. Here we represent each event by the set of actions that are being executed.

In the following modification SD' of SD, p is replaced with a non-Boolean simple fluent c with domain $\{1, \ldots, n\}$, where n is a positive number. The causal laws of SD' are

a **causes** $c = n$,

exogenous a,

inertial c.

It is easy to check that SD'_1 is the causal theory (3.6), with c_0, c_1, and a replaced by $0 : c$, $1 : c$, and $0 : a$ respectively.

4.4 A Simplified Monkey and Bananas Domain in $\mathcal{C}+$

In the example below, we will see a few more abbreviations for causal laws.

An expression of the form

$$\textbf{constraint } F$$

stands for the static law

$$\textbf{caused } \bot \textbf{ if } \neg F.$$

An expression of the form

$$\textbf{nonexecutable } a_1 \wedge \cdots \wedge a_n \textbf{ if } F, \tag{4.6}$$

where $a_1, \ldots, a_n$ are action constants, stands for the fluent dynamic law

$$\textbf{caused } \bot \textbf{ if } \top \textbf{ after } a_1 \wedge \cdots \wedge a_n \wedge F.$$

We will drop "**if** F" when $F = \top$.

See [Giunchiglia *et al.*, 2004, Appendix B] for the complete list of abbreviations for $\mathcal{C}+$ causal laws.

A $\mathcal{C}+$ action description for the Monkey and Bananas domain is shown in [Giunchiglia *et al.*, 2004, Figs. 2 and 3]. Here we consider a simplified version of this domain, which includes only the monkey and the box, but no bananas. The monkey and the box can be at one of the two locations P_1 and P_2. The monkey can walk to another place and climb onto the box, but the box cannot be moved. The action description *SMB*, shown in Figure 4.2, represents this simplified Monkey and Bananas domain in $\mathcal{C}+$.

According to the signature, there are three actions available to the monkey: $Walk(P_1)$, $Walk(P_2)$ and *ClimbOn*. According to some of the causal laws, the monkey's elevation is *Hi* (that is, the monkey is on the box) only if he is at the same location as the box; the monkey can't walk to the place where he is, and can't climb onto the box if he is already on the box; the monkey can't walk at the same time as he climbs onto the box.

The transition system represented by action description *SMB* is shown in Figure 4.3. Due to the two constraint axioms in the action description, there are only 6 states in this transition system—the elevation of the box can only be *Lo* and the monkey must be at the same location as the box whenever he is on the box. The graph consists of two disjoint parts because no action changes the location of the box.

4.5 The Causal Calculator (CCalc)

The Causal Calculator (CCalc) is a system that implements a fragment of the action language $\mathcal{C}+$. CCalc solves planning problems for action domains described

Notation: x ranges over $\{Monkey, Box\}$; p ranges over $\{P_1, P_2\}$.

Simple fluent constants:	Domains:
$Location(x)$	$\{P_1, P_2\}$
$Elevation(x)$	$\{Hi, Lo\}$

Action constants:
$Walk(p)$, $ClimbOn$

Causal laws:

constraint $Elevation(Box) = Lo$
constraint $Elevation(Monkey) = Hi \rightarrow Location(Monkey) = Location(Box)$

$Walk(p)$ **causes** $Location(Monkey) = p$
nonexecutable $Walk(p)$ **if** $Location(Monkey) = p$

$ClimbOn$ **causes** $Elevation(Monkey) = Hi$
nonexecutable $ClimbOn$ **if** $Elevation(Monkey) = Hi$

nonexecutable $Walk(p) \wedge ClimbOn$

exogenous c	for every action constant c
inertial c	for every simple fluent constant c

Figure 4.2: Action description *SMB* written in $\mathcal{C}+$.

in $\mathcal{C}+$ using ideas of satisfiability planning [Kautz and Selman, 1992].

For instance, the file shown in Figure 4.4 contains the action description *SMB* adapted to the syntactic requirements of CCalc and a query encoding a planning problem. In this planning problem, initially the monkey is at $p1$ and the box is at $p2$, and the goal is to find a plan of length 2 so that in the final state the monkey will be on the box. The output of CCalc lists the values of fluents at each time point and the actions executed between any two time points:

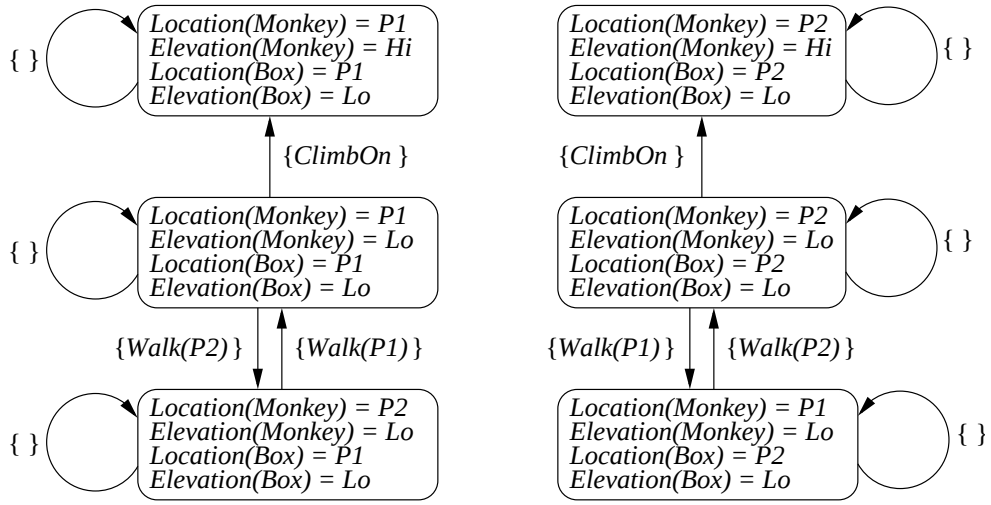

Figure 4.3: The transition system corresponding to the action description *SMB*.

```
0:  location(monkey)=p1  location(box)=p2
    elevation(monkey)=lo  elevation(box)=lo

ACTIONS:  walk(p2)

1:  location(monkey)=p2  location(box)=p2
    elevation(monkey)=lo  elevation(box)=lo

ACTIONS:  climbOn

2:  location(monkey)=p2  location(box)=p2
    elevation(monkey)=hi  elevation(box)=lo
```

CCALC has been applied to several challenging problems in the theory of

```
:- sorts
  thing;
  place;
  level.

:- objects
  monkey,box        :: thing;
  p1,p2             :: place.
  hi,lo             :: level

:- variables
  P                 :: place.

:- constants
  location(thing)       :: inertialFluent(place);
  elevation(thing)       :: inertialFluent(level);

  walk(place),
  climbOn                :: exogenousAction.

constraint elevation(box)=lo.
constraint elevation(monkey)=hi ->> location(monkey)=location(box).

walk(P) causes location(monkey)=P.
nonexecutable walk(P) if location(monkey)=P.

climbOn causes elevation(monkey)=hi.
nonexecutable climbOn if elevation(monkey)=hi.

nonexecutable walk(P) & climbOn.

:- query
maxstep :: 2;
0: location(monkey)=p1,
   location(box)=p2;
maxstep: elevation(monkey)=hi.
```

Figure 4.4: Action description *SMB* in CCALC input language, with queries

commonsense knowledge. Lifschitz *et al* [2000] represented and solved the oldest planning problem in AI—Getting to the Airport [McCarthy, 1959] using CCALC. Several enhancements of the Missionaries and Cannibals domain [Amarel, 1968] introduced in [McCarthy, 2007] were solved using CCALC in [Lifschitz, 2000]. In [Akman *et al.*, 2004] the language of CCALC was used to describe two commonsense domains of non-trivial size, the Zoo World and the Traffic World. More recently, CCALC was used for the executable specification of norm-governed computational societies [Artikis *et al.*, 2009] and for the automatic analysis of business processes under authorization constraints [Armando *et al.*, 2009].

Chapter 5

Informal Introduction to mini-MAD

The main feature of our new action description language MAD is describing actions by modules. This feature is similar to the use of subroutines in programming languages, or classes in object-oriented programming. It allows us to describe actions by referring to some related actions that were described earlier.

In this chapter, we will introduce a fragment of MAD, called mini-MAD, by examples. The syntax and semantics of mini-MAD will be described in Chapters 6 and 7.

5.1 Basic Modules: Example

It is possible to describe an action in MAD from scratch as we do in other action languages—by simply listing its effects and preconditions. This is how "abstract" descriptions of actions can be written in a general purpose library.

For instance, module MOVE (Figure 5.1) is an abstract axiomatization of

sorts
Thing;
Place;

module MOVE;

actions
Move(*Thing*,*Place*);

fluents
Location(*Thing*): *Place*;

variables
x: *Thing*; p: *Place*;

axioms
inertial *Location*(x);
exogenous *Move*(x, p);
Move(x, p) **causes** *Location*(x) = p;
nonexecutable *Move*(x, p) **if** *Location*(x) = p;

Figure 5.1: Declaration of sorts *Thing* and *Place*, and Module MOVE

"move-like" actions. We first declare two sorts: "things" and "places". The module name "MOVE" is followed by action declarations and fluent declarations: moving a thing to a place is an action, and the location of a thing is a place-valued (simple) fluent. The four axioms[1] at the end of the module are similar to causal laws of action language $\mathcal{C}+$. The location of an object is inertial: it does not change without a cause. The move actions are exogenous: they can be executed or not executed at will. Moving an object affects its location. An object cannot be moved to its current location.

Many action domains involve "move-like" actions, and describing such domains can be simplified by importing module MOVE. In the next section, we will see how this idea can be applied to the actions of walking and climbing on the box from the simplified Monkey and Bananas domain.

[1]These expressions, as well as the expressions after **axioms** in module MONKEY (Figure 5.2), are actually abbreviations; see Section 6.15.

sorts
Level;

module MONKEY;

objects
Monkey,*Box*: *Thing*;
*P*1, *P*2: *Place*;
Hi, *Lo*: *Level*;

actions
Walk(*Place*);
ClimbOn;

fluents
Elevation(*Thing*): *Level*;

variables
x: *Thing*;
p: *Place*;

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

import MOVE;
Place **is** *Level*;
Move(*Monkey*, *Hi*) **is** *ClimbOn*;
Location(*x*) **is** *Elevation*(*x*);

axioms
constraint *Elevation*(*Box*) = *Lo*;
constraint *Elevation*(*Monkey*) = *Hi* → *Location*(*Monkey*) = *Location*(*Box*);
nonexecutable *ClimbOn* ∧ *Walk*(*p*);

Figure 5.2: Declaration of sort *Level* and Module MONKEY

5.2 Importing Other Modules: Example

In mini-MAD, the simplified version of the Monkey and Bananas domain from Section 4.4 can be described by combining the sort declaration and module MOVE from Figure 5.1 with the declaration of sort *Level* and module MONKEY shown in Figure 5.2.

There are objects of three kinds: things, places and levels. *Monkey* is a thing; *P*1 and *P*2 are places; *Hi* and *Lo* are levels. The *Elevation* of a *Thing* is a simple

fluent whose value is a *Level*.

The variable declarations are followed by two import statements. Both of them import the module MOVE, but in different ways. We describe the action *Walk* in terms of *Move* by the first import statement

$$\begin{array}{l} \textbf{import}\ \text{MOVE}; \\ \quad \mathit{Move}(\mathit{Monkey}, p)\ \textbf{is}\ \mathit{Walk}(p); \end{array} \tag{5.1}$$

Intuitively, this import statement says that the action $\mathit{Walk}(p)$ has all properties that are postulated for the action $\mathit{Move}(\mathit{Monkey}, p)$ in the module MOVE. In other words, including this import statement has the same effect as including the following modification of the axioms of module MOVE:

inertial $\mathit{Location}(\mathit{Monkey})$;
exogenous $\mathit{Walk}(p)$;
$\mathit{Walk}(p)$ **causes** $\mathit{Location}(\mathit{Monkey}) = p$;
nonexecutable $\mathit{Walk}(p)$ **if** $\mathit{Location}(\mathit{Monkey}) = p$.

In addition, including this import statement has the effect of including the declaration

fluents
$\mathit{Location}(\mathit{Thing})$: Place;

from module MOVE in module MONKEY. Unlike *Location*, constant *Move* from module MOVE is not accessible in module MONKEY, because it is "renamed" in (5.1). The variables declared in module MOVE are not accessible in module MONKEY either, because variables are "local" to a module.

The second import statement

$$\begin{array}{l}\textbf{import}\ \text{MOVE};\\ \qquad \mathit{Place}\ \textbf{is}\ \mathit{Level};\\ \qquad \mathit{Move}(\mathit{Monkey}, \mathit{Hi})\ \textbf{is}\ \mathit{ClimbOn};\\ \qquad \mathit{Location}(x)\ \textbf{is}\ \mathit{Elevation}(x);\end{array} \tag{5.2}$$

describes moving "along the vertical axis." The action constant *Move* is reinterpreted again; the sort *Place* and the fluent constant *Location* are reinterpreted also. Including this import statement has the same effect as including the following modification of the axioms of module MOVE:

inertial $\mathit{Elevation}(\mathit{Monkey})$;
exogenous $\mathit{ClimbOn}$;
$\mathit{ClimbOn}$ **causes** $\mathit{Elevation}(\mathit{Monkey}) = \mathit{Hi}$;
nonexecutable $\mathit{ClimbOn}$ **if** $\mathit{Elevation}(\mathit{Monkey}) = \mathit{Hi}$.

Similarly, neither of the declaration of *Move* and the declaration of *Location* from module MOVE is included, because both the constants are "renamed" in this import statement.

The two import statements in Figure 5.2 show that each of the actions *Walk*, *ClimbOn* can be viewed as an instance of *Move*: walking is moving in the horizontal plane; climbing is moving vertically.

The axioms at the end of module MONKEY express the additional properties of the domain that have no counterparts in our general theory of move-like actions shown in Figure 5.1.

The result of appending Figure 5.2 to Figure 5.1 forms an action description of the simplified Monkey and Bananas domain in mini-MAD. We call this action

description *MSMB* (for modular simplified Monkey and Bananas). In comparison with the $\mathcal{C}+$ description *SMB* (Figure 4.2), module MONKEY has few axioms: many relevant facts come from the imported module MOVE.

5.3 Erdoğan's Implementation

In a related project, Erdoğan [2008] implemented a dialect of MAD. His software system, called MPARSE,[2] can convert a MAD action description into an input file of CCALC. For instance, Figures 5.3 and 5.4 below show how the mini-MAD code in Figures 5.1 and 5.2 can be adapted to the input language of MPARSE. Note that

```
sorts
  Thing; Place;

module MOVE;

  actions
    Move(Thing, Place);

  fluents
    Location(Thing): simple(Place);

  variables
    x: Thing;
    p: Place;

  axioms
    inertial Location(x);
    exogenous Move(x,p);
    Move(x,p) causes Location(x)=p;
    nonexecutable Move(x,p) if Location(x)=p;
```

Figure 5.3: MPARSE input file corresponding to Figure 5.1

[2] http://www.cs.utexas.edu/users/tag/mad

```
sorts
  Level;

module MONKEY;

  objects
    Monkey, Box: Thing;
    P1, P2: Place;
    Hi, Lo: Level;

  actions
    Walk(Place);
    ClimbOn;

  fluents
    Elevation(Thing): simple(Level);

  variables
    x: Thing;
    p: Place;
    l: Level;

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

  import MOVE;
    Place is Level;
    Move(Monkey,Hi) is ClimbOn;
    Location(x) is Elevation(x);

  axioms
    constraint Elevation(Box)=Lo;
    constraint Elevation(Monkey)=Hi
               -> Location(Monkey)=Location(Box);
    nonexecutable Walk(p) & ClimbOn;
```

Figure 5.4: MPARSE input file corresponding to Figure 5.2

the line

$$Location(Thing): \ Place;$$

in Figure 5.1 turns into

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

in Figure 5.3, and the declaration of *Elevation* is modified in a similar way. This is because in mini-MAD all fluents are simple, but we distinguish between simple and statically determined fluents (Section 4.1) both in the full MAD and in the input language of MPARSE.

MPARSE converts these input files into equivalent CCALC code. MPARSE and CCALC, jointly, can solve planning problems for action domains described in MAD.

Chapter 6

Syntax of mini-MAD

6.1 Names: Generalized Identifiers

An *identifier* is a string of letters, digits and underscores that begins with a letter. The following words are reserved and may not appear as identifiers:

> actions, after, axioms, Boolean, causes, constraint, exogenous, false, fluents, if, import, inertial, is, module, nonexecutable, objects, simple, sorts, staticallyDetermined, true, variables.

For technical reasons that will become clear in the following chapter, we will use symbols that are more general than identifiers, called names. A *name* is an identifier possibly preceded by several strings of the form "Im.", where m is a positive integer ('I' is the first letter of the word "import"):

<*name*> ::= {'I'<*positive integer*>'.'}* <*identifier*>

6.2 Action Descriptions

An action description is a series of sort declaration sections and modules that ends with a module:

<*action description*> ::= <*component*>* <*module*>

<*component*> ::= <*sort declaration section*> | <*module*>

For instance, action description *MSMB* (Section 5.2) has four components: a sort declaration section for *Thing* and *Place*, module MOVE, a sort declaration section for *Level*, and module MONKEY.

6.3 Sort Declaration Section

A sort declaration section has the form

sorts

s_1;

$\vdots$

s_n;

where each s_i is a name. The abstract syntax[1] is:

<*sort declaration section*> ::= <*sort name*>$^+$

<*sort name*> ::= <*name*>

6.4 Modules

A MAD module has the form

[1] In an abstract EBNF definition, some terminal symbols are omitted, as, for instance, **sorts** in the definition of <*sort declaration*>.

module *module-name*;

module-body

The *module-body* may contain several sections in the following order: an object declaration section, an action constant declaration section, a fluent constant declaration section, a variable declaration section, and a section of axioms. In addition, the *module-body* may contain any number of import statements before or after these sections. The abstract syntax is:

$<$*module*$>$ ::= $<$*module name*$>$ $<$*import statement*$>^*$
[$<$*object declaration section*$>$] $<$*import statement*$>^*$
[$<$*action declaration section*$>$] $<$*import statement*$>^*$
[$<$*fluent declaration section*$>$] $<$*import statement*$>^*$
[$<$*variable declaration section*$>$] $<$*import statement*$>^*$
[$<$*axiom section*$>$] $<$*import statement*$>^*$

$<$*module name*$>$ ::= $<$*name*$>$

A *module-name* may not be used in an action description more than once.

6.5 Object Declaration Section

An object declaration section has the form

objects

o_spec_1;

$\vdots$

o_spec_n;

where each o_spec_i is an object specification, that is, an expression of the form

$o_1, \ldots, o_m : S$

where each o_i is a name, and S is a sort name. The abstract syntax is:

<*object declaration*> ::= <*object specification*>$^+$

<*object specification*> ::= <*object name*>$^+$ <*sort name*>

<*object name*> ::= <*name*>

6.6 Action Declaration Section

An action declaration section has the form

actions

a_1;

$\vdots$

a_n;

where each a_i is a name, possibly followed by a parenthesized list of sort names (meant to specify the domains of the arguments). The abstract syntax is:

<*action declaration*> ::= <*action schema*>$^+$

<*action schema*> ::= <*action name*> <*sort name*>*

<*action name*> ::= <*name*>

6.7 Fluent Declaration Section

A fluent declaration section has the form

fluents

f_spec_1;

$\vdots$

f_spec_n;

where each f_spec_i is a fluent constant specification. A fluent constant specification has the form

$$f_1, \ldots, f_m : range$$

where each f_i is a name, possibly followed by a parenthesized list of sort names (meant to specify the domains of the arguments), and *range* is a sort name or the keyword **Boolean** (meant to specify the sort of the value). The abstract syntax is:

<*fluent declaration*> ::= <*fluent specification*>$^+$

<*fluent specification*> ::= <*Boolean spec*> | <*non-Boolean spec*>

<*Boolean spec*> ::= <*Boolean schema*>$^+$ **Boolean**

<*Boolean schema*> ::= <*Boolean fluent name*> <*sort name*>*

<*non-Boolean spec*> ::= <*non-Boolean schema*>$^+$ <*sort name*>

<*non-Boolean schema*> ::= <*non-Boolean fluent name*> <*sort name*>*

<*Boolean fluent name*> ::= <*name*>

<*non-Boolean fluent name*> ::= <*name*>

6.8 Variable Declaration Section

A variable declaration section has the form

variables

v_spec_1;

$\vdots$

v_spec_n;

where each v_spec_i is a variable specification, that is, an expression which has the form

$$v_1, \ldots, v_m : S$$

where each v_i is a name, and S is a sort name. The abstract syntax is:

<*variable declaration*> ::= <*variable specification*>$^+$

<*variable specification*> ::= <*variable name*>$^+$ <*sort name*>

<*variable name*> ::= <*name*>

Each variable name is local to the module in which its specification is.

6.9 Formulas

Axioms and import statements are constructed by formulas, with some keywords, in certain forms. Before we show the syntax of axioms and import statements below, first we define what we mean by a valid formula.

6.9.1 Terms

A *term* is an object, a variable, or a non-Boolean fluent name possibly followed by a parenthesized list of arguments:

<*term*> ::= <*simple term*> | <*non-Boolean expression*>

<*simple term*> ::= <*object name*> | <*variable name*>

<*non-Boolean expression*> ::= <*non-Boolean fluent name*> <*simple term*>*

For instance, in module MONKEY of action description *MSMB*, the expressions *Monkey*, p, *Location*(x), and *Elevation*(*Box*) are terms.

6.9.2 Boolean Expressions

A *Boolean expression* is an action constant or a Boolean fluent constant possibly followed by a parenthesized list of arguments:

<*Boolean expression*> ::= <*Boolean constant*> <*simple term*>*

<*Boolean constant*> ::= <*action name*> | <*Boolean fluent name*>

For instance, in module MONKEY of action description *MSMB*, $Move(Monkey, p)$ is a Boolean expression.

6.9.3 Atoms

There are four kinds of atoms:

<*atom*> ::= $\bot$ | $\top$ | <*Boolean expression*> | <*term*>'='<*term*>

For instance, in module MONKEY of action description *MSMB*, *ClimbOn* and $Elevation(Box) = Lo$ are atoms.

6.9.4 Formulas

A formula is built from atoms using propositional connectives:

<*formula*> ::= <*atom*> | $\neg$ <*formula*> |

<*formula*> <*binary connective*> <*formula*>

<*binary connective*> ::= $\wedge$ | $\vee$ | $\rightarrow$ | $\leftrightarrow$

For instance, the expression

$$ClimbOn \wedge Walk(p)$$

in module MONKEY of action description *MSMB* is a formula, which is a conjunction of two atom formulas.

As in $\mathcal{C}+$ (Section 4.2), 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. For instance, $\bot$, $x = Monkey$,

$Elevation(Box) = Lo$ are fluent formulas; $Move(Monkey, p)$ and $ClimbOn \wedge Walk(p)$ are action formulas; formula $Location(Monkey) = p \wedge ClimbOn$ is neither.

6.10 Axiom Section

An axiom section has the form

axioms

$axiom_1$;

$\vdots$

$axiom_n$;

where each $axiom_i$ is an expression that has the form similar to a causal law in $\mathcal{C}+$, with the main difference that in mini-MAD we drop the leading keyword **caused**:

<*axiom*> ::= <*static law*> | <*action dynamic law*> | <*fluent dynamic law*>

<*static law*> ::= <*formula*> **if** <*formula*>

<*action dynamic law*> ::= <*formula*> **if** <*formula*>

<*fluent dynamic law*> ::= <*formula*> **if** <*formula*> **after** <*formula*>

The formula in front of **if** is called the *head* of an an axiom. In a static law, both the formulas should be fluent formulas. In an action dynamic law, the head should be an action formula. In a fluent dynamic law, the head and and the formula following **if** should be fluent formulas.

Axioms in module MOVE and module MONKEY are actually abbreviations (see Section 6.15 below). For instance,

$$Move(x, p) \textbf{ causes } Location(x) = p$$

stands for the fluent dynamic law

$$Location(x) = p \textbf{ if } \top \textbf{ after } Move(x, p).$$

6.11 Import Statements

An import statement has the form

import *module-name*;
sort-renaming-clause$_1$;
$\vdots$
sort-renaming-clause$_n$;
constant-renaming-clause$_1$;
$\vdots$
constant-renaming-clause$_m$;

where $n, m \geq 0$. The abstract syntax is:

$<$*import statement*$>$ ::= $<$*module name*$>$
$<$*sort renaming clause*$>^*$
$<$*constant renaming clause*$>^*$

For instance, the second import statement in module MONKEY (Figure 5.2) contains one sort renaming clause and two constant renaming clauses.

The module named with *module-name* is called the *importee* module of this import statement. The module that contains this import statement is called the *importer* module. For any import statement in an action description, the importee module must be listed in front of the importer module. Additional conditions for an import statement will be discussed in Section 6.14.

6.11.1 Sort Renaming Clauses

A sort renaming clause has the form

$$s_1 \textbf{ is } s_2 \tag{6.1}$$

where s_1 and s_2 are sort names. The abstract syntax is:

<sort renaming clause> ::= *<sort name>* *<sort name>*

A sort name may not appear as the left-hand side of more than one sort renaming clauses in the same import statement.

6.11.2 Constant Renaming Clauses

A constant renaming clause has the form

const-renaming-lhs **is** *const-renaming-rhs* .

There are two kinds of constant renaming clauses: Boolean and non-Boolean. In a Boolean constant renaming clause, the left-hand side *const-renaming-lhs* is a Boolean constant name possibly followed by a parenthesized list of arguments and the right-hand side *const-renaming-rhs* is a formula. In a non-Boolean constant renaming clause each of *const-renaming-lhs* and *const-renaming-rhs* is a non-Boolean fluent constant possibly followed by a parenthesized list of arguments; the arguments of *const-renaming-lhs* must be distinct variables. The abstract syntax is:

<constant renaming clause> ::= *<Boolean renaming clause>* |
<non-Boolean renaming clause>

<Boolean renaming clause> ::= *<Boolean renaming lhs>* *<Boolean renaming rhs>*

<Boolean renaming lhs> ::= *<Boolean constant>* *<simple term>**

<*Boolean renaming rhs*> ::= <*formula*>

<*non-Boolean renaming clause*> ::= <*non-Boolean renaming lhs*>
<*non-Boolean renaming rhs*>

<*non-Boolean renaming lhs*> ::= <*non-Boolean fluent name*> <*variable name*>*

<*non-Boolean renaming rhs*> ::= <*non-Boolean expression*>

For instance, in the first import statement in module MONKEY (Figure 5.2),

$$Move(Monkey, p) \textbf{ is } Walk(p) \tag{6.2}$$

is a Boolean constant renaming clause, in which *Move* is a Boolean constant and $Walk(p)$ is a formula. In the second import statement in module MONKEY, the constant renaming clause

$$Location(x) \textbf{ is } Elevation(x) \tag{6.3}$$

is a non-Boolean constant renaming clause.

For a constant renaming clause, every variable occurring in the right-hand side must be one of the variables occurring in the left-hand side. A constant name may not appear on the left-hand side of more than one sort renaming clauses in the same import statement. Additional conditions will be discussed in the next section.[2]

[2]Our syntax of constant renaming clauses is somewhat different from that of the input language of MPARSE [Erdoğan, 2008]. The latter does not allow complex formulas in the right-hand side of constant renaming clauses, but it includes an additional "case construct". It also allows the right-hand side to include variables not in the left-hand side.

6.12 Interpreting Declarations

As in many other languages, whenever we refer to a name in a mini-MAD action description, the name should be declared before it is used, and it should be used in accordance with its declaration. But due to the modularity of mini-MAD, defining the precise meaning of a declaration is not as simple as, say, in the input language of CCalc.

A name can be declared not only "explicitly"—by a declaration section, but also "implicitly"—by an import statement. For instance, in module MONKEY (Figure 5.2), the name *Walk* is explicitly declared to be an action constant with an argument of sort *Place*. The name *Location* is implicitly declared, by the first import statement, to be a *Place*-valued fluent constant with an argument of sort *Thing*. Since sorts are declared outside modules and variables are local to a module, only objects and constants can be implicitly declared.

We define the precise meaning of declarations in mini-MAD below. First note that an import statement has the form

$$\begin{array}{l} \textbf{import } \mathit{NAME}; \\ \quad \mathit{sort}'_1 \textbf{ is } \mathit{sort}_1; \\ \quad \vdots \\ \quad \mathit{sort}'_k \textbf{ is } \mathit{sort}_k; \\ \quad \mathit{const}'_1 \ \cdots \textbf{ is } F_1; \\ \quad \vdots \\ \quad \mathit{const}'_l \ \cdots \textbf{ is } F_l; \end{array} \tag{6.4}$$

where *NAME* is the name of the importee module, $\mathit{sort}_1, \ldots, \mathit{sort}_k, \mathit{sort}'_1, \ldots, \mathit{sort}'_k$ are sort names, and $\mathit{const}'_1, \ldots, \mathit{const}'_l$ are constant names. The dots after each const'_j

represent the possible parenthesized arguments.

If *IS* is an import statement (6.4), and s is a sort name, we define

$$IS(s) = \begin{cases} sort_i, & \text{if } s = sort'_i \text{ for some } i \in \{1, \ldots, k\}, \\ s, & \text{otherwise.} \end{cases}$$

For instance, if *IS* is the second import statement in module MONKEY (Figure 5.2), then $IS(Thing) = Thing$, and $IS(Place) = Level$.

6.12.1 Declarations of Sort Names and Variable Names

A sort declaration section declares every sort name occurring in it. As defined above, sort declaration sections are outside modules.

The variable declaration section of a module M declares a variable name v to be of sort s if it contains a variable specification of the form

$$\ldots, v, \ldots \ : \ s;$$

6.12.2 Declarations of Object Names

About an object declaration section we say that it declares an object name o to be of sort s if it contains an object specification of the form

$$\ldots, o, \ldots \ : \ s;$$

The relation "an occurrence of an import statement *IS* in an action description declares an object name o to be of sort s" is defined recursively, as follows: with *IS* written as (6.4), this relation holds if in module *NAME* the object declaration section or an import statement declares o to be of some sort s' such that $s = IS(s')$.

6.12.3 Declarations of Action Names

About an action declaration section we say that it declares an action name a to have arguments of sorts $s_1, \ldots, s_n$ if it contains an action schema $a(s_1, \ldots, s_n)$.

The relation "an occurrence of an import statement *IS* in an action description declares an action name a to have arguments of sorts $s_1, \ldots, s_n$" is defined recursively, as follows: with *IS* written as (6.4), this relation holds if

- a is different from $const'_1, \ldots, const'_l$, and
- in module *NAME* the action declaration section or an import statement declares a to have arguments of some sorts $s'_1, \ldots, s'_n$ such that $s_1 = IS(s'_1), \ldots, s_n = IS(s'_n)$.

6.12.4 Declarations of Boolean Fluent Names

About a fluent declaration section we say that it declares a Boolean fluent name f to have arguments of sorts $s_1, \ldots, s_n$ if it contains a Boolean fluent schema $f(s_1, \ldots, s_n)$.

The relation "an occurrence of an import statement *IS* in an action description declares a Boolean fluent name f to have arguments of sorts $s_1, \ldots, s_n$" is defined recursively, as follows: with *IS* written as (6.4), this relation holds if

- f is different from $const'_1, \ldots, const'_l$, and
- in module *NAME* the fluent declaration section or an import statement declares f to have arguments of some sorts $s'_1, \ldots, s'_n$ such that $s_1 = IS(s'_1), \ldots, s_n = IS(s'_n)$.

6.12.5 Declarations of non-Boolean Fluent Names

About a fluent declaration section we say that it declares a non-Boolean fluent constant f to have arguments of sorts $s_1, \ldots, s_n$ and value of sort s if it contains a fluent constant specification of the form

$$\ldots, f(s_1, \ldots, s_n), \ldots \ : \ s;$$

The relation "an occurrence of an import statement *IS* in an action description declares a non-Boolean fluent constant f to have arguments of sorts $s_1, \ldots, s_n$ and value of sort s" is defined recursively, as follows: with *IS* written as (6.4), this relation holds if

- f is different from $const'_1, \ldots, const'_l$, and
- in module *NAME* the fluent declaration section or an import statement declares f to have arguments of some sorts $s'_1, \ldots, s'_n$ and value of some sort s' such that $s_1 = IS(s'_1), \ldots, s_n = IS(s'_n)$ and $s = IS(s')$.

For instance, the first import statement in module MONKEY (Figure 5.2) declares *Location* to have an argument of sort *Thing* and value of sort *Place*, but it doesn't declare the action name *Move*. The second import statement declares neither *Move* nor *Location*.

6.12.6 The Use of Names

With the meaning of declarations defined above, when a name u is used in a module M of a mini-MAD action description, it should be declared beforehand. More precisely,

- if u is a sort name that doesn't occur as the left-hand side of a sort renaming clause, it should be declared in a sort declaration section prior to M;
- if u is a sort name that occurs as the left-hand side of a sort renaming clause in an import statement IS, it should be declared in a sort declaration section prior to the importee module of IS;
- if u is a constant name that occurs in the left-hand side of a constant renaming clause in an import statement IS, it should be declared in the importee module of IS;
- in any other case, u should be declared earlier in M.

6.13 Multiple Declarations

In an action description, a sort name may not be declared more than once, and the same name may not be declared again by a declaration section of any module in this action description.

In a module, a name may not be declared by more than one declaration sections. In other words, in a module a name may not belong to more than one of these kinds: variable names, object names, action names, and fluent names. In addition, a declaration section may not declare the same name more than once.

Since every variable name is local to a module, each variable name v is of a unique sort s in a module M. We denote the sort s by $SORT_v^M$. The superscript M may be dropped when the context is clear.

An object name or a constant name can be declared in a module several times when import statements are present. But this is only allowed in the absence of conflicts, in the following sense:

- a name may not be declared to belong to more than one of these kinds: object names, action names, Boolean fluent names and non-Boolean fluent names;
- if an object name is declared to be of sort s and of sort s' in the same module, then $s = s'$;
- if a constant name is declared to have arguments of sorts $s_1, \ldots, s_n$ and of sorts $s'_1, \ldots, s'_m$ in the same module, then $n = m$ and $s_i = s'_i$ for every $i \in \{1, \ldots, n\}$;
- if a non-Boolean fluent name is declared to have value of sort s and of sort s' in the same module, then $s = s'$.

If u is an object name declared in module M, by $SORT_u^M$ we denote the sort of u in M; in view of the conditions above, this sort is uniquely defined. Similarly, if u is a non-Boolean fluent name, by $SORT_u^M$ we denote the sort of the value of u in M. If u is a constant name, by $SORT_{u(i)}^M$ we denote the sort of the i^{th} argument of u in M. The superscript M in these notations may be omitted if the context is clear.

For instance, the non-Boolean fluent name *Location* is declared by the fluent declaration section in module MOVE, and it is also declared by the first import statement in module MONKEY. In each of the modules,

$$SORT_{Location(1)} = Thing, \quad \text{and} \quad SORT_{Location} = Place.$$

6.14 Sort Matching Conditions

For any occurrence of a term or a Boolean expression of the form

$$c(arg_1, \ldots, arg_n) \tag{6.5}$$

that is not the left-hand side of a constant renaming clause, $SORT_{arg_i}$ must be the same as $SORT_{c(i)}$.

When (6.5) is the left-hand side of a constant renaming clause, the sort matching condition is more complicated. First, c is declared in the importee module[3] while $arg_1, \ldots, arg_n$ are declared in the importer module. Second, the import statement may include sort renaming clauses that need to be taken into account.

For any constant renaming clause with the left-hand side (6.5) in an import statement IS with an importee module M, the sort $SORT_{arg_i}$ must be the same as $IS(SORT^M_{c(i)})$. In addition, for any non-Boolean renaming clause

$$c \ldots \textbf{ is } c' \ldots; \tag{6.6}$$

the sort $SORT_{c'}$ must be the same as $IS(SORT^M_c)$.

For instance, in module MONKEY, the second import statement IS_2 with importee module MOVE contains a Boolean constant renaming clause

$$Move(Monkey, Hi) \textbf{ is } ClimbOn;$$

in accordance with the condition above,

$$SORT_{Monkey} = Thing, \quad SORT_{Hi} = Level,$$

and

$$IS_2(SORT^{MOVE}_{Move(1)}) = IS_2(Thing) = Thing,$$
$$IS_2(SORT^{MOVE}_{Move(2)}) = IS_2(Place) = Level.$$

[3]As defined above, c is not declared by the import statement, thus it is generally not declared in the importer module. Consequently $SORT_{c(i)}$ may be meaningless in the importer module, or it may have a different meaning.

As another example, IS_2 contains a non-Boolean constant renaming clause

$$Location(x) \textbf{ is } Elevation(x);$$

in accordance with the condition above,

$$SORT_{Elevation} = Level,$$

and

$$IS_2(SORT^{MOVE}_{Location}) = IS_2(Place) = Level.$$

6.15 Abbreviations

As we can see in action description *MSMB*, we have a few abbreviation forms for axioms in mini-MAD. They are similar to those in $\mathcal{C}+$ [Giunchiglia *et al.*, 2004, Appendix B]. Below is a list of abbreviations for axioms in mini-MAD.

6.15.1 causes

The expression

$$F_1 \textbf{ causes } F_2 \textbf{ if } F_3,$$

where F_1 is an action formula and F_2 is a fluent formula, stands for the fluent dynamic law

$$F_2 \textbf{ if } \top \textbf{ after } F_1 \wedge F_3.$$

The part **if** F_3 may be dropped when $F_3 = \top$. For instance, in the axiom section of module MOVE the expression

$$Move(x, p) \textbf{ causes } Location(x) = p$$

stands for the fluent dynamic law

$$Location(x) = p \textbf{ if } \top \textbf{ after } Move(x, p).$$

6.15.2 exogenous

The expression

$$\textbf{exogenous } a,$$

where a is an action name possibly followed by parenthesized arguments, stands for two action dynamic laws

$$a \textbf{ if } a$$

and

$$\neg a \textbf{ if } \neg a.$$

For instance, in the axiom section of module MOVE the expression

$$\textbf{exogenous } Move(x, p)$$

stands for two action dynamic laws

$$Move(x, p) \textbf{ if } Move(x, p)$$

and

$$\neg Move(x, p) \textbf{ if } \neg Move(x, p).$$

6.15.3 inertial

The expression

$$\textbf{inertial } f, \tag{6.7}$$

where f is a Boolean fluent name possibly followed by parenthesized arguments, stands for two fluent dynamic laws

$$f \textbf{ if } f \textbf{ after } f$$

and

$$\neg f \textbf{ if } \neg f \textbf{ after } \neg f.$$

When f in expression (6.7) is a non-Boolean fluent name, to eliminate this abbreviation we first need to declare a new variable y by adding a variable specification

$$y : \ SORT_f$$

to the variable declaration section. Then expression (6.7) stands for the fluent dynamic law

$$f = y \textbf{ if } f = y \textbf{ after } f = y.$$

For instance, in the axiom section of module MOVE the expression

$$\textbf{inertial } Location(x)$$

stands for the fluent dynamic law

$$Location(x) = p1 \textbf{ if } Location(x) = p1 \textbf{ after } Location(x) = p1;$$

where $p1$ is a new variable of sort $SORT_{Location}$, i.e., $Place$. The variable specification

$$p1 : \ Place$$

is added to the variable declaration section of module MOVE.

6.15.4 nonexecutable

$$\textbf{nonexecutable } F_1 \textbf{ if } F_2,$$

where F_1 is an action formula, stands for the fluent dynamic law

$$\bot \textbf{ after } F_1 \wedge F_2.$$

The part **if** F_2 may be dropped when $F_2 = \top$. For instance, in the axiom section of module MOVE the expression

$$\textbf{nonexecutable } Move(x, p) \textbf{ if } Location(x) = p$$

stands for the fluent dynamic law

$$\bot \textbf{ after } Move(x, p) \wedge Location(x) = p.$$

6.15.5 constraint

An expression of the form

$$\textbf{constraint } F,$$

where F is a fluent formula, stands for the static law

$$\bot \textbf{ if } \neg F.$$

For instance, in the axiom section of module MOVE the expression

$$\textbf{constraint } \mathit{Elevation}(\mathit{Box}) = \mathit{Lo}$$

stands for the static law

$$\bot \textbf{ if } \neg(\mathit{Elevation}(\mathit{Box}) = \mathit{Lo}).$$

Chapter 7

Semantics of mini-MAD

The semantics of mini-MAD is based on grounding. The semantics of $\mathcal{C}+$ action descriptions is described in [Giunchiglia *et al.*, 2004, Sections 4.2, 4.4] and is reviewed in Chapter 4.

As a preliminary step, we define a function δ that convert an arbitrary mini-MAD action description to a "formal form": a series of sort declarations followed by a single module (see Section 7.1). For instance, the single-module action description corresponding to action description *MSMB* (Figures 5.1 and 5.2) is shown in Figure 7.5. Note the names beginning with I1 and I2 in this module. Intuitively, I1.*Move* and I2.*Move* are the "copies" of the constant *Move* from module MOVE that correspond to the two import statements in module MONKEY.

Then we show how to translate any single-module action description into $\mathcal{C}+$ by grounding (see Section 7.2).

7.1 Generating a Single-module Action Description

Given an action description D in mini-MAD that contains n modules, we want to obtain a single-module action description δD corresponding to D.

The definition of δ uses the following auxiliary functions. Informally, the function α turns a module that does not contain any import statements into its "specialized form" in accordance with a given import statement. It uses "copies" of the constants declared in the module, with names containing strings of the form "Im.". Its definition uses an auxiliary function ν. The function β "merges" two modules. The function γ eliminates the first import statement from an action description.

7.1.1 Function ν: Creating an Instance of a Module Regarding to a Renaming Clause

Sort Renaming Clause

If RC is a sort renaming clause

$$s \textbf{ is } s'$$

and M is a module without import statements, by $\nu(RC, M)$ we denote the module obtained from M by replacing every occurrence of sort name s with s'. For instance, if RC is

$$\textit{Place} \textbf{ is } \textit{Level}; \tag{7.1}$$

and M is module MOVE, then $\nu(RC, M)$ is the module shown in Figure 7.1.

module MOVE;

actions
Move(*Thing*,*Level*);

fluents
Location(*Thing*): *Level*;

variables
x: *Thing*; p: *Level*;

axioms
inertial *Location*(x);
exogenous *Move*(x, p);
Move(x, p) **causes** *Location*(x)=p;
nonexecutable *Move*(x, p) **if** *Location*(x)=p;

Figure 7.1: The result of applying ν to sort renaming clause (7.1) and module MOVE (Figure 5.1)

Constant Renaming Clause

To define $\nu(RC, M)$ when RC is a constant renaming clause, we first define two auxiliary functions *Equiv* and *Cond* that will help us turn any constant renaming clause into an axiom with an equivalence as its head.[1]

Any constant renaming clause RC can be written in the form

$$c(arg_1, \ldots, arg_n) \textbf{ is } rhs(arg\_var), \tag{7.2}$$

where each of $arg_1, \ldots, arg_n$ is a variable or an object, and arg_var is the list of variables among $arg_1, \ldots, arg_n$. Note that, if RC is a non-Boolean renaming clause, then arg_i is a variable for all i, and arg_var is the list $arg_1, \ldots, arg_n$.

[1] These equivalences are similar to rules for lifting statements to other contexts in [McCarthy, 1993], and to $\mathcal{C}+$ bridge rules studied in [Erdoğan and Lifschitz, 2006].

If RC is a Boolean renaming clause, by $Equiv(RC)$ we denote the formula

$$c(x_1, \ldots, x_n) \leftrightarrow rhs(x\_var), \tag{7.3}$$

where $x_1, \ldots, x_n$ are new identifiers, and x_var is the list of x_i's for all i such that arg_i is a variable.

If RC is a non-Boolean renaming clause, by $Equiv(RC)$ we denote the formula

$$c(x_1, \ldots, x_n) = y \leftrightarrow rhs(x_1, \ldots, x_n) = y, \tag{7.4}$$

where $x_1, \ldots, x_n, y$ are new identifiers.

For any constant renaming clause RC, by $Cond(RC)$ we denote the formula

$$\bigwedge_i (x_i = arg_i),$$

where the conjunction extends over all i such that arg_i is an object. It is clear that $Cond(RC)$ is $\top$ if RC is a non-Boolean renaming clause since none of these arg_i is an object.

The axiom that corresponds to RC, as we will see below, has the form

$$Equiv(RC) \textbf{ if } Cond(RC).$$

For instance, if RC is

$$Move(Monkey, p) \textbf{ is } Walk(p), \tag{7.5}$$

then arg_var is p (the only variable among $Monkey$ and p). For new identifiers x_1

and x_2, x_var is x_2 since only arg_2 is a variable. $Equiv(RC)$ is

$$Move(x_1, x_2) \leftrightarrow Walk(x_2),$$

and $Cond(RC)$ is

$$x_1 = Monkey.$$

As another example, if RC is

$$Location(x) \textbf{ is } Elevation(x),$$

then $Equiv(RC)$ is

$$Location(x_1) = y \leftrightarrow Elevation(x_1) = y.$$

With the above auxiliary definitions, for any constant renaming clause RC of form (7.2) and any module M without import statements, $\nu(RC, M)$ is defined as the module obtained from M by

- adding variable specifications

$$x_i \ : \ SORT_{c(i)} \qquad (1 \leq i \leq n)$$

  to the variable declaration section,

- adding the variable specification

$$y \ : \ SORT_c$$

to the variable declaration section if RC is a non-Boolean renaming clause,

- inserting the axiom

$$Equiv(RC) \textbf{ if } Cond(RC) \tag{7.6}$$

at the beginning of the axiom section,

- inserting the axiom

$$\neg c(x_1, \ldots, x_n) \textbf{ if } \neg Cond(RC) \tag{7.7}$$

at the beginning of the axiom section if RC is a Boolean renaming clause.[2]

For instance, if M is the module shown in Figure 7.1, and RC is

$$Move(Monkey,\ Hi) \textbf{ is } ClimbOn; \tag{7.8}$$

then $\nu(RC,M)$ is the module shown in Figure 7.2. The second axiom, of form (7.6), expresses that the action $Move(Monkey, Hi)$ is synonymous with $ClimbOn$. The first axiom, of form (7.7), expresses that $Move(x1, p1)$ is **false** if $x1$ or $p1$ is any other value. For instance, $Move(Box, Hi)$ is never executable.

If we apply ν again to the non-Boolean constant renaming clause

$$Location(x) \textbf{ is } Elevation(x); \tag{7.9}$$

and the module in Figure 7.2, we will get Figure 7.3.

[2]Strictly speaking, $\nu(RC, M)$ may not be a module, in the sense that some symbols occurring in $Equiv(RC)$ and $Cond(RC)$ come from RC and they are not declared in M.

module MOVE;

actions
Move(*Thing*,*Level*);

fluents
Location(*Thing*): *Level*;

variables
x, $x1$: *Thing*;
p, $p1$: *Level*;

axioms
$\neg Move(x1, p1)$ **if** $\neg((x1 = Monkey) \wedge (p1 = Hi))$;
$Move(x1, p1) \leftrightarrow ClimbOn$ **if** $(x1 = Monkey) \wedge (p1 = Hi)$;
inertial $Location(x)$;
exogenous $Move(x, p)$;
$Move(x, p)$ **causes** $Location(x) = p$;
nonexecutable $Move(x, p)$ **if** $Location(x) = p$;

Figure 7.2: The result of applying ν to Boolean constant renaming clause (7.8) and the module shown in Figure 7.1

7.1.2 Function α: Creating an Instance of a Module Regarding to an Import Statement

On the level of abstract syntax, an import statement has the form

$$NAME\ RC_1 \cdots RC_k, \tag{7.10}$$

where *NAME* is a module name (denoting the name of the importee module) and each RC_i is either a sort renaming clause or a constant renaming clause (see Section 6.11).

Let M be a module without import statements, *IS* an import statement (7.10) such that *NAME* is the name of M, and m a positive integer. For every i such that RC_i is a constant renaming clause, RC_i begins with a constant name, we will denote

module MOVE;

actions
Move(*Thing*,*Level*);

fluents
Location(*Thing*): *Level*;

variables
x, $x1$, $x2$: *Thing*;
p, $p1$, $p2$: *Level*;

axioms
$Location(x2) = p2 \leftrightarrow Elevation(x2) = p2$ **if** $\top$;
$\neg Move(x1, p1)$ **if** $\neg((x1 = Monkey) \wedge (p1 = Hi))$;
$Move(x1, p1) \leftrightarrow ClimbOn$ **if** $(x1 = Monkey) \wedge (p1 = Hi)$;
inertial $Location(x)$;
exogenous $Move(x, p)$;
$Move(x, p)$ **causes** $Location(x) = p$;
nonexecutable $Move(x, p)$ **if** $Location(x) = p$;

Figure 7.3: The result of applying ν to non-Boolean constant renaming clause (7.9) and the module shown in Figure 7.2

this constant name by c_i.

By $\alpha(M, IS, m)$ we denote the module obtained from

$$\nu(RC_k, \nu(RC_{k-1}, \ldots, \nu(RC_1, M) \cdots)$$

by prepending "Im." to every occurrence of every variable name and to every occurrence of each constant name c_i.

For instance, if M is module MOVE, IS is the second import statement (5.2) in module MONKEY, and $m = 2$, then $\alpha(M, IS, m)$ is shown in Figure 7.4. it is obtained from the module in Figure 7.3 by prepending "I2." to every occurrence of every variable name and to every occurrence of each of the constant names *Move* and *Location*.

module MOVE;

actions
I2.*Move*(*Thing*,*Level*);

fluents
I2.*Location*(*Thing*): *Level*;

variables
I2.x, I2.$x1$, I2.$x2$: *Thing*;
I2.p, I2.$p1$, I2.$p2$: *Level*;

axioms
I2.*Location*(I2.$x2$) = I2.$p2$ $\leftrightarrow$ *Elevation*(I2.$x2$) = I2.$p2$ **if** $\top$;
$\neg$I2.*Move*(I2.$x1$, I2.$p1$) **if** $\neg((\text{I2}.x1 = \mathit{Monkey}) \wedge (\text{I2}.p1 = \mathit{Hi}))$;
I2.*Move*(I2.$x1$, I2.$p1$) $\leftrightarrow$ *ClimbOn* **if** $(\text{I2}.x1 = \mathit{Monkey}) \wedge (\text{I2}.p1 = \mathit{Hi})$;

inertial I2.*Location*(I2.x);
exogenous I2.*Move*(I2.x, I2.p);
I2.*Move*(I2.x, I2.p) **causes** I2.*Location*(I2.x) = I2.p;
nonexecutable I2.*Move*(I2.x, I2.p) **if** I2.*Location*(I2.x) = I2.p;

Figure 7.4: The result of applying α to module MOVE (Figure 5.1) and import statement (5.2), with $m = 2$

7.1.3 Function β: Merging Two Modules

For any module M and any module M' that does not contain any import statements, $\beta(M, M')$ is the module obtained from M by appending the contents of each section of M' (object specifications, the action schemas, fluent specifications, variable specifications, and axioms) to the corresponding section of M, with repetitions removed. Note that module $\beta(M, M')$ preserves the name of module M.

7.1.4 Function γ: Eliminating an Import Statement

For a mini-MAD action description D containing at least one import statement, we will define an action description $\gamma(D)$ that contains fewer import statements than D.

Let M be the first module in D containing an import statement, *IS* be the

first import statement in M, and M' be the importee module of IS. By $\gamma(D)$ we denote the action description obtained from D by replacing M with

$$\beta(M^*, \alpha(M', IS, m)),$$

where M^* is the module obtained from M by dropping IS, and m is the smallest positive integer such that the string "Im." does not occur in D.

It is clear that applying γ to an action description decrements the number of import statements by 1.

Recall that in a mini-MAD action description a module can only import modules before it. This condition guarantees that M' does not contain any import statements, so that it satisfies the requirement to be the first argument of α in the definition of γ.

7.1.5 Function δ: Generating a Single-module Action Description

If D contains p import statements then $\gamma^p(D)$ is an action description that does not contain any import statements. By δD we denote the result of dropping all the modules but the last one from $\gamma^p(D)$. δD is the single-module action description that we consider to have the same meaning as D.

For instance, $\delta\, MSMB$ is shown in Figure 7.5.

7.2 Grounding

For any mini-MAD action description D, we will define the corresponding $\mathcal{C}+$ action description, denoted by $cplus(D)$.

sorts
Thing; *Place*;
sorts
Level;

module MONKEY;

objects
Monkey, *Box*: *Thing*;
*P*1, *P*2: *Place*;
Hi, *Lo*: *Level*;

actions
Walk(*Place*); *ClimbOn*;
I1.*Move*(*Thing*,*Place*); I2.*Move*(*Thing*,*Level*);

fluents
Location(*Thing*): *Place*;
Elevation(*Thing*): *Level*;
I2.*Location*(*Thing*): *Level*;

variables
x: *Thing*; *p*: *Place*;
I1.*x*, I1.*x*1: *Thing*; I1.*p*, I1.*p*1: *Place*;
I2.*x*, I2.*x*1, I2.*x*2: *Thing*; I2.*p*, I2.*p*1, I2.*p*2: *Level*;

axioms
constraint $Elevation(Box) = Lo$
constraint $Elevation(Monkey) = Hi \rightarrow Location(Monkey) = Location(Box)$
nonexecutable $ClimbOn \wedge Walk(p)$;

$\neg$I1.*Move*(I1.*x*1, I1.*p*1) **if** $\neg$(I1.*x*1 = *Monkey*);
I1.*Move*(I1.*x*1, I1.*p*1) $\leftrightarrow$ *Walk*(I1.*p*1) **if** I1.*x*1 = *Monkey*;
inertial *Location*(I1.*x*);
exogenous I1.*Move*(I1.*x*, I1.*p*);
I1.*Move*(I1.*x*, I1.*p*) **causes** *Location*(I1.*x*) = I1.*p*;
nonexecutable I1.*Move*(I1.*x*, I1.*p*) **if** *Location*(I1.*x*) = I1.*p*;

I2.*Location*(I2.*x*2) = I2.*p*2 $\leftrightarrow$ *Elevation*(I2.*x*2) = I2.*p*2 **if** $\top$;
$\neg$I2.*Move*(I2.*x*1, I2.*p*1) **if** $\neg$((I2.*x*1 = *Monkey*) $\wedge$ (I2.*p*1 = *Hi*));
I2.*Move*(I2.*x*1, I2.*p*1) $\leftrightarrow$ *ClimbOn* **if** (I2.*x*1 = *Monkey*) $\wedge$ (I2.*p*1 = *Hi*);
inertial I2.*Location*(I2.*x*);
exogenous I2.*Move*(I2.*x*, I2.*p*);
I2.*Move*(I2.*x*, I2.*p*) **causes** I2.*Location*(I2.*x*) = I2.*p*;
nonexecutable I2.*Move*(I2.*x*, I2.*p*) **if** I2.*Location*(I2.*x*) = I2.*p*;

Figure 7.5: δ *MSMB*: a single module action description corresponding to *MSMB*

7.2.1 Universe of a Sort

Let D be a mini-MAD action description, and let s be a sort name declared in D. The *universe* of s, denoted by $U_D(s)$, is the set of all objects names o such that o is declared to be of sort s in the module of δD. For instance, if D is the action description *MSMB* (Figures 5.1 and 5.2), δD is shown in Figure 7.5, and the universes of the sorts *Thing*, *Place* and *Level* are

$$\begin{aligned} &U_{MSMB}(Thing) = \{Monkey, Box\}, \\ &U_{MSMB}(Place) = \{P1, P2\}, \\ &U_{MSMB}(Level) = \{Hi, Lo\}. \end{aligned} \tag{7.11}$$

Let D be a mini-MAD action description such that each set $U_D(s)$ is non-empty. We obtain $cplus(D)$ by constructing its signature (Section 7.2.2) and its causal laws (Section 7.2.3).

7.2.2 Signature of $cplus(D)$

The signature σ of $cplus(D)$ is determined by the action and fluent declaration sections of δD, as follows. If the action declaration section contains an action schema $a(s_1, \ldots, s_k)$, then σ includes the symbols $a(z_1, \ldots, z_k)$ for all objects $z_1 \in U_D(s_1), \ldots, z_k \in U_D(s_k)$; they are designated as action constants. If the fluent declaration section contains a fluent specification

$$\ldots, c(s_1, \ldots, s_k), \cdots : s$$

then σ includes the symbols $c(z_1, \ldots, z_k)$ for all $z_1 \in U_D(s_1), \ldots, z_k \in U_D(s_k)$; they are designated as simple fluent constants with the domain $U_D(s)$ if s is a sort name,

and simple Boolean fluent constants if s is **Boolean**.

For instance, if D is *MSMB* as shown in Figures 5.1 and 5.2, then σ consists of the action constants

$$\begin{array}{l} Walk(P1), \quad Walk(P2), \quad ClimbOn, \\ \text{I1}.Move(Monkey, P1), \quad \text{I1}.Move(Monkey, P2), \\ \text{I1}.Move(Box, P1), \quad \text{I1}.Move(Box, P2), \\ \text{I2}.Move(Monkey, Hi), \quad \text{I2}.Move(Monkey, Lo), \\ \text{I2}.Move(Box, Hi), \quad \text{I2}.Move(Box, Lo), \end{array}$$

the simple fluent constants

$$Location(Monkey), \quad Location(Box)$$

with domain $\{P1, P2\}$, and the simple fluent constants

$$Elevation(Monkey), \quad Elevation(Box), \quad \text{I2}.Location(Monkey), \quad \text{I2}.Location(Box)$$

with domain $\{Hi, Lo\}$.

7.2.3 Causal Laws of $cplus(D)$

The causal laws of $cplus(D)$ are obtained from the axioms of δD by the following procedure:

1. "Ground" each axiom by substituting arbitrary elements of $U_D(SORT_v^{\delta D})$ for every variable v in that axiom.

2. Replace each atom of the form c, where c is a Boolean fluent name possibly followed by parenthesized arguments, by $c = \mathbf{true}$.

3. Replace each atom of the form $o_1 = o_2$, where o_1 and o_2 are object names, by $\top$ if o_1 and o_2 are the same, and by $\bot$ otherwise.

4. Replace each atom of the form $c_1 = c_2$, where c_1 and c_2 are non-Boolean fluent names possibly followed by parenthesized arguments, by $\bigvee_{o \in \Gamma} (c_1 = o \wedge c_2 = o)$, where $\Gamma = U_D(SORT_{c_1}) \cap U_D(SORT_{c_2})$.

5. Replace each atom of the form $o = c$, where o is an object name and c is a non-Boolean fluent name possibly followed by parenthesized arguments, by $c = o$.

6. Replace each atom of the form $c = o$, where c is a non-Boolean fluent name possibly followed by parenthesized arguments and o is an object name such that $SORT_c \neq SORT_o$, by $\bot$.

7. Prepend the keyword **caused** to each axiom.

For instance, in δ *MSMB* (Figure 7.5) the abbreviation

constraint $Elevation(Monkey) = Hi \rightarrow Location(Monkey) = Location(Box)$

stands for the static law

$\bot$ **if** $\neg(Elevation(Monkey) = Hi \rightarrow Location(Monkey) = Location(Box))$.

Steps 1, 2 and 3 don't apply; in step 4 it is translated to

$\bot$ **if** $\neg(Elevation(Monkey) = Hi$
$\rightarrow \bigvee_{o \in \{P1,P2\}}(Location(Monkey) = o \wedge Location(Box) = o))$.

Steps 5 and 6 don't apply; finally after step 7, we get the corresponding causal law in $cplus(MSMB)$:

$$\textbf{caused } \bot \textbf{ if } \neg(Elevation(Monkey) = Hi \rightarrow \bigvee_{o \in \{P1,P2\}}(Location(Monkey) = o \wedge Location(Box) = o)).$$

As another example, the axiom

$$\text{I2}.Move(\text{I2}.x1, \text{I2}.p1) \leftrightarrow ClimbOn \textbf{ if } (\text{I2}.x1 = Monkey) \wedge (\text{I2}.p1 = Hi)$$

in $\delta\, MSMB$ is turned into four axioms by step 1:

$$\text{I2}.Move(Monkey, Hi) \leftrightarrow ClimbOn \textbf{ if } (Monkey = Monkey) \wedge (Hi = Hi),$$
$$\text{I2}.Move(Monkey, Lo) \leftrightarrow ClimbOn \textbf{ if } (Monkey = Monkey) \wedge (Lo = Hi),$$
$$\text{I2}.Move(Box, Hi) \leftrightarrow ClimbOn \textbf{ if } (Box = Monkey) \wedge (Hi = Hi),$$
$$\text{I2}.Move(Box, Lo) \leftrightarrow ClimbOn \textbf{ if } (Box = Monkey) \wedge (Lo = Hi).$$

Steps 2 and 3 turn these axioms into

$$\text{I2}.Move(Monkey, Hi) \leftrightarrow (ClimbOn = \textbf{true}) \textbf{ if } \top \wedge \top,$$
$$\text{I2}.Move(Monkey, Lo) \leftrightarrow (ClimbOn = \textbf{true}) \textbf{ if } \top \wedge \bot,$$
$$\text{I2}.Move(Box, Hi) \leftrightarrow (ClimbOn = \textbf{true}) \textbf{ if } \bot \wedge \top,$$
$$\text{I2}.Move(Box, Lo) \leftrightarrow (ClimbOn = \textbf{true}) \textbf{ if } \bot \wedge \bot.$$

Steps 4, 5 and 6 don't apply, and step 7 prepends **caused** to each of these expressions.

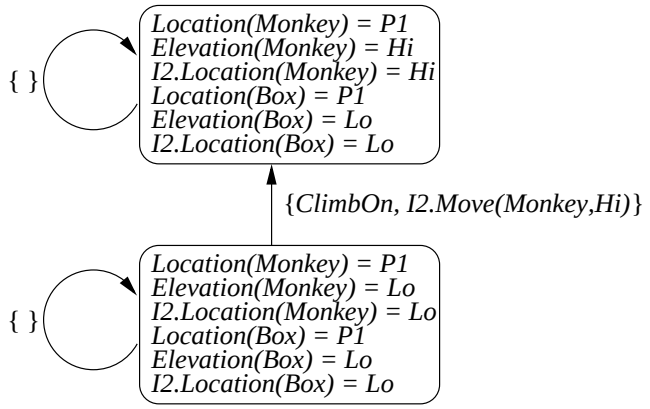

Figure 7.6: A part of the transition system represented by action description δ *SMB*.

7.2.4 States and Transitions

According to the semantics of mini-MAD, the transition system represented by a mini-MAD action description D is the transition system represented by the $\mathcal{C}+$ action description $cplus(D)$.

For instance, the transition system represented by action description *MSMB* is the directed graph shown in Figure 4.3, but with different labels, in view of the presence of the constants beginning with I1 and I2. Figure 7.6 shows the counterpart of the edge in the North-West corner of Figure 4.3. As discussed in Section 4.1, an event that includes more than one actions is usually understood as the concurrent execution of these actions. The intuitive meaning of the event in Figure 7.6 is different: *ClimbOn* and I2.$Move(Monkey, Hi)$ are not concurrently executed actions; these expressions describe the same action in two different ways.

Chapter 8

Syntax of MAD

8.1 Additional Features of the Full Language

In this chapter we extend the syntax of mini-MAD (Chapter 6) by a few useful constructs.

First, statically determined fluents will be now added.

Second, we will be able to say that one sort is a subsort of another. For example, in a MAD description of the Logistics domain [Veloso, 1992], we will be able to state that the sorts *Truck* and *Plane* are subsorts of the sort *Vehicle*.

Third, quantifiers will be available in formulas, not only propositional connectives. For instance, we will be able to express that a vehicle cannot be driven if no one is inside by writing

nonexecutable $Drive(v)$ **if** $\neg\,\exists x\ In(x, v)$.

Fourth, the syntax of MAD will allow us to use nested constants—a constant as an argument of another constant. For instance, we will be able to express that

the *Monkey* is walking to the current location of the *Box* by the formula

$$Walk(Location(Box)). \tag{8.1}$$

This is more concise and intuitive than the equivalent formula

$$\exists p(Location(Box) = p \wedge Walk(p)). \tag{8.2}$$

Finally, sort names may be used in MAD as unary predicates.

The semantics of MAD does not use grounding, because some of the extensions mentioned above would make the definition of grounding significantly more complicated.

The definition of the universe of a sort (Section 7.2.1) would become more complex in the presence of subsorts. For instance, the universe of the sort *Vehicle* should include all objects of sorts *Truck* and *Plane*, and all objects of subsorts of these sorts. (See the definition of $|s|$ in Section 11.1.)

In addition to substituting objects for free variables (Section 7.2.3), we would have to eliminate quantifiers in favor of multiple conjunctions and disjunctions.

Prior to grounding, formulas containing nested constants would have to be eliminated by a syntactic transformation that would turn, say, (8.1) to (8.2).

For all these reasons, and also because future extensions of MAD are likely to lead to similar complications, we chose not to use grounding in the semantics of the full language. Instead, its semantics is based on first-order causal logic, reviewed in Chapter 9.

8.2 Extending the Syntax of mini-MAD

An action description in MAD has the same structure as in mini-MAD (Section 6.2): it is a series of sort declaration sections and modules that ends with a module.

The syntax of sort declaration sections and modules is as defined in Sections 6.3 and 6.4. A module still is an ordered series of sections: an object declaration section, an action declaration section, a fluent declaration section, a variable declaration section and an axiom section, with any number of import statements before or after any section mentioned above.

The syntax of object declaration sections, action declaration sections, variable declaration sections and import statements is as defined in Sections 6.5, 6.6, 6.8, and 6.11.

The syntax of fluent declaration sections and axiom sections, as well as the syntax of formulas, will be generalized.

8.2.1 Fluent Declaration Section

The syntax of fluent declaration sections is changed to include statically determined fluents. In the abstract syntax shown in Section 6.7, the definitions of <*Boolean spec*> and <*non-Boolean spec*> are replaced by the following rules:

<*Boolean spec*> ::= <*Boolean schema*>$^+$<*fluent kind*>

<*non-Boolean spec*> ::= <*non-Boolean schema*>$^+$<*fluent kind*>‘(’<*sort name*>‘)’

<*fluent kind*> ::= **simple** | **staticallyDetermined**

For instance, in the syntax of MAD, the declaration of non-Boolean simple fluent *Location* in module MOVE would be

fluents

Location(*Thing*): **simple**(*Place*);

as in the input language of MPARSE (Section 5.3).

Otherwise, the syntax of MAD is compatible with the syntax of mini-MAD: any mini-MAD action description D will become a MAD action description if we replace every fluent specification

$$f_1, \ldots, f_m \ : \ range$$

by

$$f_1, \ldots, f_m \ : \ \textbf{simple}(range),$$

and drop "(*range*)" when *range* is **Boolean**. In the future, we will disregard this difference and identify any mini-MAD action description with the corresponding MAD action description.

8.2.2 Formulas

The syntax of formulas in mini-MAD (Section 6.9.4) is generalized to allow the use of quantifiers:

<*formula*> ::= <*atom*> | ¬ <*formula*> |
<*formula*> <*binary connective*> <*formula*> |
<*quantifier*> <*variable name*> <*formula*>

<*quantifier*> ::= ∀ | ∃

The syntax of atoms is defined as in Section 6.9.3, while the definition of Boolean expressions in mini-MAD is generalized to include nested constants and

sort predicates. In the abstract syntax shown in Section 6.9.2, the definition of <*Boolean expression*> is replaced by

<*Boolean expression*> ::= <*Boolean constant*> <*term*>* |
<*sort name*> '(' <*term*> ')'

The definition of terms is generalized to include nested constants: in the abstract syntax shown in Section 6.9.1, the definition of <*non-Boolean expression*> is replaced by

<*non-Boolean expression*> ::= <*non-Boolean fluent name*> <*term*>*

8.2.3 Axiom Section

The syntax of axiom sections is generalized to include sort inclusion expressions. In the abstract syntax shown in Section 6.10, the definition of <*axiom*> is replaced by the following rules:

<*axiom*> ::= <*static law*> | <*action dynamic law*> | <*fluent dynamic law*>
| <*sort inclusion expression*>

<*sort inclusion expression*> ::= < *sort name* > "<<" <*sort name*>

For instance,

Truck << *Vehicle*;

is an axiom.

No sort name is allowed in the head of any axiom of the first three kinds; no statically determined fluent name is allowed in the head of a fluent dynamic law (this is similar to the requirement in $\mathcal{C}+$, see Section 4.2). Without either of these restrictions, the theorems in Chapter 11 would become incorrect.

8.3 Interpreting Declarations

Declarations in MAD are interpreted in the same way as in mini-MAD (Section 6.12). The condition on multiple declarations of a name and the definitions of $SORT_u^M$ and $SORT_{u(i)}^M$ (Section 6.13) remain the same.

8.4 Sort Matching Conditions

As in mini-MAD, for any non-Boolean renaming clause

$$c(arg_1, \ldots, arg_n) \textbf{ is } \cdots$$

in an import statement IS with importee module M, we require that $SORT_{arg_i}$ be equal to $IS(SORT_{c(i)}^M)$.

Otherwise, the sort matching conditions from Section 6.14 do not apply to the full language. Requiring the sort of an argument of a constant c always match the declaration of c would be too restrictive in the presence of subsorts. For instance, in the example from Section 8.1 we would not be able to use a term of the subsort *Truck* of *Vehicle* as the argument of *Drive*. Even the (intuitively meaningless) expression

$$Location(Hi) \tag{8.3}$$

in the context of the module MONKEY would be considered a syntactically correct MAD formula. The semantics of mini-MAD, based on a translation into $\mathcal{C}+$ (Section 7.2), would not be applicable to expression (8.3), because (8.3) is not among the contants of the signature of $cplus(MSMB)$. In MAD, on the other hand, the idea that an argument of an expression should be of appropriate sort will be expressed

semantically (Section 10.2.2).

8.5 Example

Consider the action description *RDB* shown in Figure 8.1. This action description contains two modules: CARRY and DELIVER. Module CARRY describes general "carry-like" actions. Prior to this module, we declare three sorts. In this module the action *Carry* is declared to have three arguments of sorts *Agent*, *Thing* and *Place* respectively, and the fluent *Location* is declared in the same way as in module MOVE. The first axiom is a sort inclusion expression, and it states that sort *Agent* is a subsort of sort *Thing*. The next four axioms are similar to those in module MOVE. The last axiom says that action *Carry* cannot be executed if the agent and the thing are not at the same location. After module CARRY and the declaration of a new sort *Block*, there is the module DELIVER. In this module, another sort inclusion expression is present, stating that sort *Block* is a subsort of *Thing* also. The import statement at the end of module DELIVER says, intuitively, action *Deliver* has all the properties of action *Carry* if we limit the first argument of *Carry* to be the specific object *Robot* and limit the second argument to be of sort *Block*—a subsort of $SORT_{Carry(2)}$, i.e., *Thing*. Note that in the import statement, the expression $Carry(Robot, b, p)$ would not be allowed in mini-MAD according to the sort matching conditions.

sorts
Agent; *Thing*; *Place*;

module CARRY;

actions
Carry(*Agent*, *Thing*, *Place*);

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

variables
g: *Agent*; x: *Thing*; p: *Place*;

axioms
Agent << *Thing*;

exogenous $Carry(g, x, p)$;
inertial $Location(x)$;

$Carry(g, x, p)$ **causes** $Location(x) = p \wedge Location(g) = p$;

nonexecutable $Carry(g, x, p)$ **if** $Location(x) = p$;
nonexecutable $Carry(g, x, p)$ **if** $\neg(Location(g) = Location(x))$;

sorts
Block;

module DELIVER;

objects
Robot: *Agent*;
$B1$, $B2$: *Block*;
$P1$, $P2$: *Place*;

actions
Deliver(*Block*, *Place*);

variables
b: *Block*; p: *Place*;

axioms
Block << *Thing*;

import CARRY;
$Carry(Robot, b, p)$ **is** $Deliver(b, p)$;

Figure 8.1: Action description *RDB*

Chapter 9

Review of First-order Causal Logic

9.1 Syntax and Semantics of First-order Causal Theories

First-order causal logic is the basis of our semantics of MAD (Chapter 10). The review of the syntax and semantics of first-order causal theories in this section follows [Lifschitz, 1997, Section 2]. As in the propositional version of causal logic reviewed in Section 3.1, the main idea of nonmonotonic causal logic [McCain and Turner, 1997] is to distinguish between the claim that a proposition is true and the stronger claim that there is a *cause* for it to be true. Causal dependencies are described by *causal rules*—expressions of the form

$$F \Leftarrow G, \tag{9.1}$$

where F and G are first-order formulas. Rule (9.1) expresses that there is a cause for the head formula F to hold if the body formula G holds, or, in other words, that G provides a "causal explanation" for F.

A (first-order) *causal theory* is defined by

- a finite subset of the signature[1] of the underlying language, called the *explainable symbols* of the theory, and
- a finite set of causal rules.

In the definition of the semantics of causal theories below, we use the substitution of variables for the explainable symbols in a formula. In connection with this, it is convenient to denote formulas by expressions like $F(E)$, where E is the list of all explainable symbols. Then, for any tuple e of variables that is similar[2] to E, the result of replacing all occurrences of the constants E in $F(E)$ by the variables e can be denoted by $F(e)$.

Consider a causal theory T with the explainable symbols E and the causal rules

$$F_i(E, x^i) \Leftarrow G_i(E, x^i) \qquad (i = 1, \dots),$$

where x^i is the list of all free variables of the i-th rule. Take a tuple e of new variables similar to E. By $T^*(e)$ we denote the formula

$$\bigwedge_i \forall x^i (G_i(E, x^i) \to F_i(e, x^i)).$$

[1] The *signature* of a first-order language is the set of its function constants and predicate constants (other than equality). This includes, in particular, object constants (function constants of arity 0) and propositional constants (predicate constants of arity 0).

[2] The similarity condition means that (i) e has the same length as E, (ii) if the k-th member of E is a function constant then the k-th member of e is a function variable of the same arity, and (iii) if the k-th member of E is a predicate constant then the k-th member of e is a predicate variable of the same arity.

Note that the occurrences of explainable symbols in the heads are replaced here by variables, and the occurrences in the bodies are not. We will view T as shorthand for the sentence

$$\forall e(T^*(e) \leftrightarrow e = E). \tag{9.2}$$

(The expression $e = E$ stands for the conjunction of the equalities between the members of e and the corresponding members of E.) For instance, by a *model* of T we mean a model of (9.2); a formula is *entailed* by T if it is entailed by (9.2). Note that the tuple e may contain function and predicate variables, so that (9.2) is, generally, a second-order formula.

Intuitively, the condition $T^*(e)$ expresses that the possible values e of the explainable symbols E are "causally explained" by the rules of T. Sentence (9.2) says that the actual values of these symbols are the only ones that are explained by the rules of T.

9.2 Example

Let T be the causal theory with the rules

$$\begin{aligned} &Block(B1) \Leftarrow \top, \\ &Block(B2) \Leftarrow \top, \\ &\neg Block(x) \Leftarrow \neg Block(x), \end{aligned} \tag{9.3}$$

where the predicate constant *Block* is explainable, and the object constants $B1$, $B2$ are not explainable. Intuitively, the last line of (9.3) expresses the closed-world assumption: if x is not a block then there is a cause for this. In this case, E is

Block, e is a unary predicate variable, and $T^*(e)$ is

$$e(B1) \wedge e(B2) \wedge \forall x(\neg \mathit{Block}(x) \rightarrow \neg e(x)).$$

The second-order sentence (9.2) is equivalent in this case to the first-order sentence

$$\forall x(\mathit{Block}(x) \leftrightarrow x = B1 \vee x = B2). \tag{9.4}$$

Chapter 10

Semantics of MAD

In this chapter, we talk about the semantics of MAD based on the first-order causal logic which is reviewed in Chapter 9. As in Chapter 7, we first turn an action description D into its "normal form" δD. Then we translate this single-module MAD action description into a causal theory in the sense of first-order causal logic.

10.1 Generating a Single-module Description

The definition of δ for MAD, as the corresponding definition for mini-MAD, uses auxiliary functions ν, α, β and γ. It differs from the definition given in Chapter 7 in that the construction of the formula $Cond(RC)$, used to define ν (Section 7.1.1), is a little different: it is modified in connection with the relaxation of sort matching conditions (Section 8.4). Let RC be a Boolean constant renaming clause of form (7.2). In MAD the formula $Cond(RC)$ is

$$G_1 \wedge \cdots \wedge G_n, \tag{10.1}$$

where G_i stands for

- $x_i = arg_i$ if arg_i is an object,
- $SORT_{arg_i}(x_i)$ if arg_i is a variable,

where $x_1, \ldots, x_n$ are new identifiers. For instance, if RC is (7.5), then $Cond(RC)$ is

$$(x_1 = Monkey) \wedge Place(x_2),$$

where x_1, x_2 are new identifiers.

On the other hand, $Cond(RC) = \top$ for non-Boolean renaming clause RC, same as in mini-MAD.

The definitions of functions ν, α, β, γ and δ are the same as those in mini-MAD (see Sections 7.1.1 to 7.1.5). Finally, δD generates a single-module MAD action description that is considered equivalent to D.

For instance, if IS is the import statement in action description RDB (Figure 8.1), RC is the constant renaming clause

$$Carry(Robot, b, p) \textbf{ is } Deliver(b, p); \tag{10.2}$$

in IS, and M is module CARRY, then in the axiom

$$Equiv(RC) \textbf{ if } Cond(RC)$$

corresponding to (10.2), the formula $Cond(RC)$ will be

$$(g1 = Robot) \wedge SORT_b(x1) \wedge SORT_p(p1),$$

module CARRY;

actions

$Carry(Agent, Thing, Place)$;

fluents

$Location(Thing)$: **simple**$(Place)$;

variables

g: $Agent$; x: $Thing$; p: $Place$;
$g1$: $Agent$; $x1$: $Thing$; $p1$: $Place$;

axioms

$\neg Carry(g1, x1, p1)$ **if** $\neg((g1 = Robot) \wedge Block(x1) \wedge Place(p1))$;
$Carry(g1, x1, p1) \leftrightarrow Deliver(x1, p1)$ **if** $(g1 = Robot) \wedge Block(x1) \wedge Place(p1)$;
$Agent << Thing$;
exogenous $Carry(g, x, p)$;
inertial $Location(x)$;
$Carry(g, x, p)$ **causes** $Location(x) = p \wedge Location(g) = p$;
nonexecutable $Carry(g, x, p)$ **if** $Location(x) = p$;
nonexecutable $Carry(g, x, p)$ **if** $\neg(Location(g) = Location(x))$;

Figure 10.1: $\nu(RC, M)$ where RC is (10.2) and M is module CARRY

that is,

$$(g1 = Robot) \wedge Block(x1) \wedge Place(p1),$$

and the formula $Equiv(RC)$ will be

$$Carry(g1, x1, p1) \leftrightarrow Deliver(x1, p1),$$

where $g1, x1, p1$ are new identifiers. The module $\nu(RC, M)$ is shown in Figure 10.1. The module $\alpha(M, IS, 1)$ can be obtained from Figure 10.1 by prepending "I1." to each occurrence of every variable name and each occurrence of action name $Carry$. The "normal form" action description δRDB is shown in Figure 10.2.

sorts
Agent; *Thing*; *Place*;

sorts
Block;

module DELIVER;

objects
Robot: *Agent*;
*B*1, *B*2: *Block*;
*P*1, *P*2: *Place*;

actions
Deliver(*Block*, *Place*);
I1.*Carry*(*Agent*, *Thing*, *Place*);

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

variables
b: *Block*; *p*: *Place*;
I1.*g*: *Agent*; I1.*x*: *Thing*; I1.*p*: *Place*;
I1.*g*1: *Agent*; I1.*x*1: *Thing*; I1.*p*1: *Place*;

axioms
Block << *Thing*;
$\neg$ I1.*Carry*(I1.*g*1, I1.*x*1, I1.*p*1) **if** $\neg$ ((I1.*g*1 = *Robot*) $\wedge$ *Block*(I1.*x*1) $\wedge$ *Place*(I1.*p*1));
I1.*Carry*(I1.*g*1, I1.*x*1, I1.*p*1) $\leftrightarrow$ *Deliver*(I1.*x*1, I1.*p*1)
if (I1.*g*1 = *Robot*) $\wedge$ *Block*(I1.*x*1) $\wedge$ *Place*(I1.*p*1);
Agent << *Thing*;
exogenous I1.*Carry*(I1.*g*, I1.*x*, I1.*p*);
inertial *Location*(I1.*x*);
I1.*Carry*(I1.*g*, I1.*x*, I1.*p*) **causes** *Location*(I1.*x*) = I1.*p* $\wedge$ *Location*(I1.*g*) = I1.*p*;
nonexecutable I1.*Carry*(I1.*g*, I1.*x*, I1.*p*) **if** *Location*(I1.*x*) = I1.*p*;
nonexecutable I1.*Carry*(I1.*g*, I1.*x*, I1.*p*) **if** $\neg$(*Location*(I1.*g*) = *Location*(I1.*x*));

Figure 10.2: Action description δRDB

10.2 Translating a Single-module MAD Action Description to a First-order Causal Theory

Since any action description in MAD can be turned into a single-module action description by applying function δ, it is sufficient that we describe the translation to a first-order causal theory only for MAD action descriptions in this "normal" form.

Given such an action description D and a nonnegative integer m (the length of the behaviors that we are interested in), the causal theory D_m is defined below.

10.2.1 The Signature of D_m

The signature σ^{D_m} consists of

- an explainable unary predicate constant s for each sort name s;
- a non-explainable object constant o for each object name o;
- the non-explainable object constant *None*;
- an explainable predicate constant $i:c$ for each action name c and every $i \in \{0,\dots,m-1\}$; the arity of $i:c$ is the same as the arity of c;
- a predicate constant $i:c$ for each Boolean fluent name c and every $i \in \{0,\dots,m\}$; the arity of $i:c$ is the same as the arity of c; this constant is explainable except for the case when c is simple and $i=0$;
- a function constant $i:c$ for each non-Boolean fluent name c and every $i \in \{0,\dots,m\}$; the arity of $i:c$ is the same as the arity of c; this constant is explainable except for the case when c is simple and $i=0$.

For instance, let D be the single-module MAD action description δRDB shown in Figure 10.2. The signature σ^{D_m} consists of the non-explainable object constants

$$\mathit{Robot},\ B1,\ B2,\ P1,\ P2,\ \mathit{None},$$

the explainable predicate constants

$$\mathit{Agent},\ \mathit{Thing},\ \mathit{Place},\ \mathit{Block},\ i{:}\mathit{Deliver},\ i{:}\mathrm{I1}.\mathit{Carry} \qquad (0 \le i < m),$$

and the unary function constants

$$i : \mathit{Location} \qquad (0 \leq i \leq m)$$

that are explainable when $i > 0$.

The prefixes $i :$ are "time stamps", as in Section 4.2. For instance, the value of

$$5 : \mathit{Location}(\mathit{Robot})$$

is the location of *Robot* at time 5; the truth value of

$$3 : \mathit{Deliver}(B2, P1)$$

shows whether the robot delivers the block $B2$ to place $P1$ between times 3 and 4.

10.2.2 The Causal Rules of D_m

In the list of the causal rules of D_m below, the following notation is used: by i we denote all integers in $\{0, \ldots, m\}$; by j we denote all integers in $\{0, \ldots, m-1\}$; for any formula F, $i{:}F$ stands for the result of prepending "i:" to all fluent names and action names in F; by $x_1, \ldots, x_n, y$ we denote distinct object variables.

The set of causal rules of D_m consists of two groups. The first group is determined by the signature δ^{D_m}, as follows:

(i) $\neg s(x) \Leftarrow \neg s(x)$ for each sort name s (the closed-world assumption for sorts);

(ii) $\mathit{SORT}_o(o) \Leftarrow \top$ for each object name o;

(iii) $o \neq None \Leftarrow \top$ for each object name o;[1]

(iv) $o_1 \neq o_2 \Leftarrow \top$ for each pair of distinct object names o_1, o_2 (the unique names assumption for objects);

(v) the rules

$$\neg\, j : c(x_1, \ldots, x_n) \Leftarrow \neg\, SORT_{c(1)}(x_1) \vee \cdots \vee \neg\, SORT_{c(n)}(x_n)$$

for each action name c of arity n (all arguments should be of appropriate sorts);

(vi) the rules

$$\neg\, i : c(x_1, \ldots, x_n) \Leftarrow \neg\, SORT_{c(1)}(x_1) \vee \cdots \vee \neg\, SORT_{c(n)}(x_n)$$

for each Boolean fluent name c of arity n, and the rules

$$i : c(x_1, \ldots, x_n) = None \Leftarrow \neg\, SORT_{c(1)}(x_1) \vee \cdots \vee \neg\, SORT_{c(n)}(x_n)$$

for each non-Boolean fluent name c of arity n (all arguments should be of appropriate sorts);

(vii) the rules

$$i : c(x_1, \ldots, x_n) \neq y \Leftarrow \neg SORT_c(y) \wedge SORT_{c(1)}(x_1) \wedge \cdots \wedge SORT_{c(n)}(x_n)$$

for each non-Boolean fluent name c of arity n (the value should be of the appropriate sort).

[1] An expression of the form $a \neq b$ is shorthand for the expression $\neg(a = b)$.

The second group of causal rules is determined by the axioms in D. It is similar to the process of translating causal laws of $\mathcal{C}+$ into propositional causal logic described in [Giunchiglia *et al.*, 2004, Section 4.2] and reviewed in Section 4.2. This group of causal rules includes:

(viii) $i : F \Leftarrow i : G \wedge \bigwedge_x SORT_x(x)$, where the conjunction extends over all variables x occurring in F or G, for each static law

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

in the list of axioms;

(ix) $j : F \Leftarrow j : G \wedge \bigwedge_x SORT_x(x)$, where the conjunction extends over all variables x occurring in F or G, for each action dynamic law

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

in the list of axioms;

(x) $j+1 : F \Leftarrow j+1 : G \wedge j : H \wedge \bigwedge_x SORT_x(x)$, where the conjunction extends over all variables x occurring in F, G or H, for each fluent dynamic law

$$F \ \textbf{if} \ G \ \textbf{after} \ H$$

in the list of axioms;

(xi) $s_1(x) \rightarrow s_2(x) \Leftarrow \top$ for each sort inclusion expression $s_1 << s_2$ in the list of axioms.

This concludes the definition of our translation from the language of action

descriptions into the language of causal theories.

For instance, the causal theory $(\delta RDB)_m$, corresponding to the single-module MAD action description δRDB (Figure 10.2), is shown in Figure 10.3.

10.3 States and Transitions

For any MAD action description D, each model of the causal theory $(\delta D)_m$ represents a possible "history" of the state-transition system described by D over the time instants $0, \ldots, m$. In particular, the models of $(\delta D)_0$ are the *states* of D, and the models of $(\delta D)_1$ are the *transitions* of D.

For instance, consider the MAD action description RDB (Figure 8.1) and the corresponding causal theory $(\delta RDB)_m$ (Figure 10.3). The signature $\sigma^{(\delta RDB)_0}$ is

$$\{Agent,\ Thing,\ Place,\ Block,\ Robot,\ B1,\ B2,\ P1,\ P2,\ None,\ 0\!:\!Location\},$$

where the first four symbols are explainable; the signature $\sigma^{(\delta RDB)_1}$ is

$$\sigma^{(\delta RDB)_0} \cup \{0\!:\!Deliver,\ 0\!:\!\mathrm{I}1.Carry,\ 1\!:\!Location\},$$

where all the three added symbols are explainable.

We will give a few examples of models of causal theories $(\delta RDB)_0$ and $(\delta RDB)_1$. In these examples, the universe U is the set

$$\{Robot,\ B1,\ B2,\ P1,\ P2,\ None\}. \tag{10.3}$$

Or, more generally, U can be any subset of (10.3).

Notation: $o, o' \in \{Robot,\ B1,\ B2,\ P1,\ P2\},\ \ i \in \{0, \ldots, m\},\ \ j \in \{0, \ldots, m-1\}.$

$$
\begin{aligned}
\neg Agent(x) &\Leftarrow \neg Agent(x),\\
\neg Thing(x) &\Leftarrow \neg Thing(x),\\
\neg Place(x) &\Leftarrow \neg Place(x),\\
\neg Block(x) &\Leftarrow \neg Block(x),
\end{aligned}
$$

$$
\begin{aligned}
Agent(Robot) &\Leftarrow \top,\\
Block(B1) &\Leftarrow \top,\\
Block(B2) &\Leftarrow \top,\\
Place(P1) &\Leftarrow \top,\\
Place(P2) &\Leftarrow \top,\\
o \neq None &\Leftarrow \top,\\
o \neq o' &\Leftarrow \top \qquad (o \neq o'),
\end{aligned}
$$

$$
\begin{aligned}
\neg\, j\!:\!Deliver(x_1, x_2) &\Leftarrow \neg Block(x_1) \vee \neg Place(x_2),\\
\neg\, j\!:\!\mathrm{I1}.Carry(x_1, x_2, x_3) &\Leftarrow \neg Agent(x_1) \vee \neg Thing(x_2) \vee \neg Place(x_3),\\
i\!:\!Location(x_1) = None &\Leftarrow \neg Thing(x_1),\\
i\!:\!Location(x_1) \neq y &\Leftarrow \neg Place(y) \wedge Thing(x_1),
\end{aligned}
$$

$$
\begin{aligned}
Block(x) \rightarrow Thing(x) &\Leftarrow \top,\\
Agent(x) \rightarrow Thing(x) &\Leftarrow \top,
\end{aligned}
$$

$$
\begin{aligned}
\neg\, j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g1, \mathrm{I1}.x1, \mathrm{I1}.p1) &\Leftarrow \neg\,((\mathrm{I1}.g1 = Robot) \wedge Block(\mathrm{I1}.x1) \wedge Place(\mathrm{I1}.p1))\\
&\qquad \wedge Agent(\mathrm{I1}.g1) \wedge Thing(\mathrm{I1}.x1) \wedge Place(\mathrm{I1}.p1),\\
j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g1, \mathrm{I1}.x1, \mathrm{I1}.p1) &\leftrightarrow j\!:\!Deliver(\mathrm{I1}.x1, \mathrm{I1}.p1)\\
&\Leftarrow (\mathrm{I1}.g1 = Robot) \wedge Block(\mathrm{I1}.x1) \wedge Place(\mathrm{I1}.p1)\\
&\qquad \wedge Agent(\mathrm{I1}.g1) \wedge Thing(\mathrm{I1}.x1) \wedge Place(\mathrm{I1}.p1),
\end{aligned}
$$

$$
\begin{aligned}
j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g, \mathrm{I1}.x, \mathrm{I1}.p) &\Leftarrow j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g, \mathrm{I1}.x, \mathrm{I1}.p)\\
&\qquad \wedge Agent(\mathrm{I1}.g) \wedge Thing(\mathrm{I1}.x) \wedge Place(\mathrm{I1}.p),\\
\neg\, j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g, \mathrm{I1}.x, \mathrm{I1}.p) &\Leftarrow \neg\, j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g, \mathrm{I1}.x, \mathrm{I1}.p)\\
&\qquad \wedge Agent(\mathrm{I1}.g) \wedge Thing(\mathrm{I1}.x) \wedge Place(\mathrm{I1}.p),
\end{aligned}
$$

$$
\begin{aligned}
j+1\!:\!Location(\mathrm{I1}.x) = \mathrm{I1}.p &\Leftarrow j+1\!:\!Location(\mathrm{I1}.x) = \mathrm{I1}.p \wedge j\!:\!Location(\mathrm{I1}.x) = \mathrm{I1}.p\\
&\qquad \wedge Place(\mathrm{I1}.p) \wedge Thing(\mathrm{I1}.x),
\end{aligned}
$$

$$
\begin{aligned}
j+1\!:\!Location(\mathrm{I1}.x) = \mathrm{I1}.p \wedge j+1\!:\!Location(\mathrm{I1}.g) = \mathrm{I1}.p\\
\Leftarrow j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g, \mathrm{I1}.x, \mathrm{I1}.p) \wedge Agent(\mathrm{I1}.g1) \wedge Place(\mathrm{I1}.p) \wedge Thing(\mathrm{I1}.x),
\end{aligned}
$$

$$
\begin{aligned}
\bot \Leftarrow\ & j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g, \mathrm{I1}.x, \mathrm{I1}.p) \wedge j\!:\!Location(\mathrm{I1}.x) = \mathrm{I1}.p\\
&\wedge Agent(\mathrm{I1}.g) \wedge Thing(\mathrm{I1}.x) \wedge Place(\mathrm{I1}.p),\\
\bot \Leftarrow\ & j\!:\!\mathrm{I1}.Carry(\mathrm{I1}.g, \mathrm{I1}.x, \mathrm{I1}.p) \wedge \neg(j\!:\!Location(\mathrm{I1}.g) = j\!:\!Location(\mathrm{I1}.x))\\
&\wedge Agent(\mathrm{I1}.g) \wedge Thing(\mathrm{I1}.x) \wedge Place(\mathrm{I1}.p).
\end{aligned}
$$

Figure 10.3: Rules of causal theory $(\delta RDB)_m$

Consider two interpretations s_0 and s_1 of $\sigma^{(\delta RDB)_0}$ that are defined by

$$s_0[Agent](x) = s_1[Agent](x) = \begin{cases} \textbf{true} & \text{if } x = Robot, \\ \textbf{false} & \text{otherwise,} \end{cases}$$

$$s_0[Thing](x) = s_1[Thing](x) = \begin{cases} \textbf{true} & \text{if } x \in \{Robot, B1, B2\}, \\ \textbf{false} & \text{otherwise,} \end{cases}$$

$$s_0[Place](x) = s_1[Place](x) = \begin{cases} \textbf{true} & \text{if } x \in \{P1, P2\}, \\ \textbf{false} & \text{otherwise,} \end{cases}$$

$$s_0[Block](x) = s_1[Block](x) = \begin{cases} \textbf{true} & \text{if } x \in \{B1, B2\}, \\ \textbf{false} & \text{otherwise,} \end{cases}$$

$$s_0[o] = s_1[o] = o \qquad o \in \{Robot, B1, B2, P1, P2, None\},$$

$$s_0[0\!:\!Location](x) = \begin{cases} P1 & \text{if } x \in \{Robot, B1, B2\}, \\ None & \text{otherwise,} \end{cases}$$

$$s_1[0\!:\!Location](x) = \begin{cases} P2 & \text{if } x \in \{Robot, B1\}, \\ P1 & \text{if } x = B2, \\ None & \text{otherwise.} \end{cases}$$

Interpretations s_0 and s_1 are two models of $(\delta RDB)_0$, that is, states of RDB.

Consider the interpretation tr of $\sigma^{(\delta RDB)_1}$ that is defined by

$$tr[c] = s_0[c],\ c \in \{Agent, Thing, Place, Block, Robot, B1, B2, P1, P2, None\},$$
$$tr[0\!:\!Location] = s_0[0\!:\!Location],$$
$$tr[1\!:\!Location] = s_1[0\!:\!Location],$$
$$tr[0\!:\!Deliver](x,p) = \begin{cases} \textbf{true} & \text{if } x = B1 \wedge p = P2, \\ \textbf{false} & \text{otherwise,} \end{cases}$$
$$tr[0\!:\!\mathrm{I}1.Carry](g,x,p) = \begin{cases} \textbf{true} & \text{if } g = Robot \wedge x = B1 \wedge p = P2, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

This interpretation tr is a model of $(\delta RDB)_1$ and consequently a transition of RDB.

Intuitively, transition tr starts at state s_0 and ends at state s_1: at time 0, both the robot and the block $B1$ are at Place $P1$; between time 0 and 1, the robot delivers the $B1$ from $P1$ to $P2$; consequently at time 1 both the robot and $B1$ are at $P2$. The extent of every sort remain unchanged during the transition.

Chapter 11

Properties of MAD

In this chapter we argue that the new approach to the semantics of MAD action descriptions with variables has some desirable, intuitively expected mathematical properties. Since the semantics of MAD uses the function δ that turns an arbitrary action description into a single-module action description, we can limit our attention, without loss of generality, to action descriptions in this normal form.

11.1 Extent of a Sort

Let D be a single-module MAD action description. Proposition 1 below shows that in any model of D_m the extent of any sort S corresponds to the set of the object names that one would expect to belong to S in accordance with the object declarations and sort inclusion expressions of D.

To make this claim precise, we will use the following notation. First we define the *sort graph* of D as follows: its vertices are the sort names $s_1, \ldots, s_n$ declared in D; its edges are the pairs (s_i, s_j) such that the sort inclusion expression $s_i << s_j$ is an axiom of D. This graph is usually acyclic. For any sort name s, by $|s|$ we denote

the set of object names o such that s is reachable from $SORT_o$ in the sort graph of D.

In the special case when D is a mini-MAD action description, the sort graph of D has no edges, and $|s|$ is the same as $U_D(s)$.

Proposition 1 *Assume that D is a single-module MAD action description. For any sort name s declared in D, the causal theory D_m entails*

$$\forall x \left(s(x) \leftrightarrow \bigvee_{o: o \in |s|} x = o \right).$$

For instance, assume that the sort declaration section of D is

sorts

Vehicle; *Plane*; *Truck*;

the object declaration section is

objects

*Car*1, *Boat*1 : *Vehicle*;

*Plane*1 : *Plane*;

*Truck*1, *Truck*2 : *Truck*;

and the sort inclusion expressions in the axiom section are

Plane << *Vehicle*;

Truck << *Vehicle*;

It is clear that $|Vehicle| = \{Car1, Boat1, Plane1, Truck1, Truck2\}$. By Proposi-

tion 1, D_m entails

$$\forall x(\mathit{Vehicle}(x) \leftrightarrow x = \mathit{Car}1 \vee x = \mathit{Boat}1 \vee x = \mathit{Plane}1 \vee x = \mathit{Truck}1 \vee x = \mathit{Truck}2). \tag{11.1}$$

The assumption that the heads of axioms of D don't contain sort names in Section 8.2.3 is essential. For instance, if the above sort declaration section also contains another sort name *Thing*, the above object declaration section also contains the object specification

$$\mathit{Helicopter1} : \mathit{Thing}$$

and the axiom section contains the axiom

$$\mathit{Vehicle}(\mathit{Helicopter1}) \textbf{ if } \top,$$

then (11.1) is not entailed by D_m although $|\mathit{Vehicle}|$ remains the same.

11.2 States and Transitions

In the theory of $\mathcal{C}+$, the view that histories of length m can be thought of as paths in a transition system is justified by two theorems, Propositions 7 and 8 from [Giunchiglia *et al.*, 2004]. The first of them shows that any transition "starts" in a state and "ends" in a state. According to the second theorem, an interpretation of the signature of D_m is a model of D_m if and only if it "consists of m transitions." Propositions 2 and 3 below are similar to these theorems.

Let D be a single-module MAD action description. For any interpretation I of σ^{D_1}, by I^0 and I^1 we denote the interpretations of σ^{D_0} defined as follows. (Here

$|I|$ stands for the universe of I.)

$$
\begin{aligned}
&|I^i| = |I|,\\
&I^i[s] = I[s] \text{ for every sort name } s,\\
&I^i[o] = I[o] \text{ for every object name } o,\\
&I^i[0:c] = I[i:c] \text{ for every fluent name } c,\\
&I^i[None] = I[None].
\end{aligned}
$$

Proposition 2 *For any transition I of D, the interpretations I^0 and I^1 are states of D.*

Intuitively, if I is a transition, then I^0 is the state at which I starts and I^1 is the state at which I ends. For instance, in the example in Section 10.3, $tr^0 = s_0$ and $tr^1 = s_1$.

For any interpretation I of σ^{D_m}, by $I^{(i)}$ $(0 \leq i < m)$ we denote the interpretations of σ^{D_1} defined as follows:

$$
\begin{aligned}
&|I^{(i)}| = |I|,\\
&I^{(i)}[s] = I[s] \text{ for every sort name } s,\\
&I^{(i)}[o] = I[o] \text{ for every object name } o,\\
&I^{(i)}[0{:}c] = I[i{:}c] \text{ for every fluent or action name } c,\\
&I^{(i)}[1{:}c] = I[i+1{:}c] \text{ for every fluent name } c,\\
&I^{(i)}[None] = I[None].
\end{aligned}
$$

Proposition 3 *For any positive integer m and any interpretation I of σ^{D_m}, I is a model of D_m iff every $I^{(i)}$ $(0 \leq i < m)$ is a transition of D.*

Intuitively, for any "history" I of length m, $I^{(i)}$ is the i-th transition of that history.

11.3 Proofs

The proofs of these propositions refer to properties of circumscription [McCarthy, 1980, McCarthy, 1986], and we will begin with the review of this concept.

11.3.1 Review of Circumscription

The review of circumscription follows [Lifschitz, 1994].

First some notation is introduced. For any predicate symbols P, Q of the same arity,

$$P = Q \text{ stands for } \forall \mathbf{x}(P(\mathbf{x}) \leftrightarrow Q(\mathbf{x})),$$
$$P \leq Q \text{ stands for } \forall \mathbf{x}(P(\mathbf{x}) \rightarrow Q(\mathbf{x})),$$

where $\mathbf{x}$ is a tuple of distinct variables. Furthermore,

$$P < Q \text{ stands for } (P \leq Q) \wedge \neg(P = Q).$$

Let $A(P)$ be a sentence containing a predicate constant P. The *circumscription* of P in $A(P)$ (denoted by CIRC$[A(P); P]$) is the second-order sentence

$$A(P) \wedge \neg \exists p[A(p) \wedge p < P],$$

where p is a predicate variable.

Let $\mathbf{P}$ and $\mathbf{Q}$ be tuples $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_n$ of predicate symbols such that, for each $i = 1, \ldots, n$, Q_i has the same arity as P_i. Then $\mathbf{P} = \mathbf{Q}$ stands for

$$P_1 = Q_1 \wedge \cdots \wedge P_n = Q_n,$$

$\mathbf{P} \leq \mathbf{Q}$ stands for

$$P_1 \leq Q_1 \wedge \cdots \wedge P_n \leq Q_n,$$

and $\mathbf{P} < \mathbf{Q}$ is

$$(\mathbf{P} \leq \mathbf{Q}) \wedge \neg(\mathbf{P} = \mathbf{Q}).$$

Then the *parallel circumscription* of the tuple $\mathbf{P}$ in $A(\mathbf{P})$, CIRC$[A(\mathbf{P}); \mathbf{P}]$, is defined as shorthand for the expression

$$A(\mathbf{P}) \wedge \neg\exists\mathbf{p}[A(\mathbf{p}) \wedge \mathbf{p} < \mathbf{P}],$$

where $\mathbf{p}$ is the tuple $p_1, \ldots, p_n$ of distinct predicate variables with appropriate arities.

A formula is *definite* in the predicates $\mathbf{P}$ (that is, $P_1, \ldots, P_n$) if it is the conjunction of implications of the form

$$F(\mathbf{P}) \rightarrow P_i(t_1, \ldots, t_k),$$

where $t_1, \ldots, t_k$ are terms and $F(\mathbf{P})$ is a formula such that every occurrence of each P_i in it is positive. (An occurrence of an atom in a formula is positive if it is in the antecedent of an even number of implications.)

A *predicate expression* is of the form

$$\lambda x_1 \cdots, x_n F(x_1, \ldots, x_n). \tag{11.2}$$

Predicate expressions are convenient for describing formulas obtained by substitution. If E is (11.2), and $t_1, \ldots, t_n$ are terms, then $E(t_1, \ldots, t_n)$ stands for the formula $F(t_1, \ldots, t_n)$. If $A(P)$ is a formula containing a predicate constant P, and E is a predicate expression of the same arity as P, then $A(E)$ stands for the result

of replacing each atomic part $P(t_1, \ldots, t_n)$ in $A(P)$ by $E(t_1, \ldots, t_n)$ (after renaming the bound variables in A in the usual way, if necessary). For instance, if $A(P)$ is $P(a) \wedge P(b)$, then $A(\lambda x(x = y))$ is $a = y \wedge b = y$. For a tuple $\mathbf{E}$ of predicate expressions, $A(\mathbf{E})$ is defined in a similar way corresponding to $A(\mathbf{P})$.

In the statement of Lemma 1 [Lifschitz, 1994, Section 3.5], $A(\mathbf{P})$ is the universal closure of a formula that is definite in $\mathbf{P}$, and $\mathbf{E}$ is the tuple $E_1, \ldots, E_n$, where E_i denotes the predicate expression

$$\lambda \mathbf{x} \forall \mathbf{p}(A(\mathbf{p}) \rightarrow p_i(\mathbf{x})) \qquad (i = 1, \ldots, n).$$

Lemma 1 *The sentence* $A(\mathbf{E})$ *is universally valid.*

11.3.2 Lemmas

Besides Lemma 1, the following lemmas are used in the proofs of Propositions 1 to 3. In the statement of Lemma 2, the symbols $\mathbf{P}$, $A(\mathbf{P})$ and $\mathbf{E}$ have the same meaning as in the statement of Lemma 1, described in Section 11.3.1 above.

Lemma 2 *The circumscription* $\mathrm{CIRC}[A(\mathbf{P}); \mathbf{P}]$ *entails* $\mathbf{E} = \mathbf{P}$.

Proof

$$\begin{aligned} \mathrm{CIRC}[A(\mathbf{P}); \mathbf{P}] &= A(\mathbf{P}) \wedge \neg \exists \mathbf{p}(A(\mathbf{p}) \wedge \mathbf{p} < \mathbf{P}) \\ &\Leftrightarrow A(\mathbf{P}) \wedge \forall \mathbf{p}(\neg A(\mathbf{p}) \vee \neg(\mathbf{p} < \mathbf{P})) \\ &\Leftrightarrow A(\mathbf{P}) \wedge \forall \mathbf{p}(\neg A(\mathbf{p}) \vee \neg(\mathbf{p} \leq \mathbf{P}) \vee \mathbf{p} = \mathbf{P}). \end{aligned}$$

The second conjunctive term entails

$$\neg A(\mathbf{E}) \vee \neg(\mathbf{E} \leq \mathbf{P}) \vee (\mathbf{E} = \mathbf{P}).$$

By Lemma 1, $A(\mathbf{E})$ is universally valid. Consequently

$$\mathrm{CIRC}[A(\mathbf{P}); \mathbf{P}] \models \neg(\mathbf{E} \leq \mathbf{P}) \vee (\mathbf{E} = \mathbf{P}).$$

To complete the proof of this lemma, we will show that $\mathrm{CIRC}[A(\mathbf{P}); \mathbf{P}] \models \mathbf{E} \leq \mathbf{P}$, in other words, for any i, $\mathrm{CIRC}[A(\mathbf{P}); \mathbf{P}] \models E_i \leq P_i$.

Since $E_i(\mathbf{x}) = \forall \mathbf{p}(A(\mathbf{p}) \rightarrow p_i(\mathbf{x}))$, the formula

$$\forall \mathbf{x}[E_i(\mathbf{x}) \rightarrow (A(\mathbf{P}) \rightarrow P_i(\mathbf{x}))]$$

is logically valid. Since $\mathrm{CIRC}[A(\mathbf{P}); \mathbf{P}] \models A(\mathbf{P})$, it follows that

$$\mathrm{CIRC}[A(\mathbf{P}); \mathbf{P}] \models \forall \mathbf{x}[E_i(\mathbf{x}) \rightarrow P_i(\mathbf{x})].$$

The last formula is $E_i \leq P_i$. ■

Lemma 3 is a generalization of Proposition 1 from [Lifschitz, 1997]:

Lemma 3 *A causal theory of the form*

$$\begin{aligned} F_i &\Leftarrow \quad (1 \leq i \leq m), \\ \neg P_j(\mathbf{x}) &\Leftarrow \neg P_j(\mathbf{x}) \quad (1 \leq j \leq n), \end{aligned}$$

where predicate constants $P_1, \ldots, P_n$ are the only explainable symbols of this theory and each $\mathbf{x}$ is a tuple of distinct variables, is equivalent to the parallel circumscription of $P_1, \ldots, P_n$ in $\bigwedge_{i=1}^{m} \widetilde{\forall} F_i$.[1]

Proof Denote the given causal theory by T, and let $F(\mathbf{P})$ stand for $\bigwedge_{i=1}^{m} \widetilde{\forall} F_i$, where

[1] By $\widetilde{\forall} F$ we denote the universal closure of F.

$\mathbf{P}$ denotes the tuple of $P_1, \ldots, P_n$. Then

$$\begin{array}{rcl} T^*(\mathbf{p}) & \Leftrightarrow & F(\mathbf{p}) \wedge \bigwedge_{j=1}^{n} \forall \mathbf{x}(\neg P_j(\mathbf{x}) \rightarrow \neg p_j(\mathbf{x})) \\ & \Leftrightarrow & F(\mathbf{p}) \wedge (\mathbf{p} \leq \mathbf{P}), \end{array}$$

and consequently

$$\begin{array}{rcl} T & = & T^*(\mathbf{P}) \wedge \forall \mathbf{p}(T^*(\mathbf{p}) \rightarrow \mathbf{p} = \mathbf{P}) \\ & \Leftrightarrow & F(\mathbf{P}) \wedge \forall \mathbf{p}(F(\mathbf{p}) \wedge (\mathbf{p} \leq \mathbf{P}) \rightarrow \mathbf{p} = \mathbf{P}) \\ & \Leftrightarrow & F(\mathbf{P}) \wedge \forall \mathbf{p}(\mathbf{p} < \mathbf{P} \rightarrow \neg F(\mathbf{p})) \\ & \Leftrightarrow & F(\mathbf{P}) \wedge \neg \exists \mathbf{p}(F(\mathbf{p}) \wedge \mathbf{p} < \mathbf{P}) \\ & = & \mathrm{CIRC}[F(\mathbf{P}); \mathbf{P}]. \quad \blacksquare \end{array}$$

Lemma 4 below is the reproduction of Lemma 3 in [Lifschitz, 1997].

About causal theories T_1, T_2 with sets E_1, E_2 of explainable symbols we say that they are *disjoint* if

- E_1 and E_2 are disjoint sets,
- the symbols in E_1 do not occur in the heads of the rules of T_2, and the symbols in E_2 do not occur in the heads of the rules of T_1.

For any pairwise disjoint causal theories $T_1, \ldots, T_m$, define their *union* to be the causal theory obtained by combining their rules and their explainable symbols.

Lemma 4 *The union of pairwise disjoint causal theories $T_1, \ldots, T_m$ is equivalent to the conjunction $T_1 \wedge \cdots \wedge T_m$.*

Lemma 5 *Let T be a causal theory with the set E of explainable symbols. If T' is a causal theory such that*

- *its set E' of explainable symbols is a subset of E,*
- *its set of causal rules is a subset of T,*
- *every causal rule of T containing a symbol from E' in the head belongs to T',*

then $T \models T'$.

Proof Denote $E \setminus E'$ by E'', Let T'' be the causal theory consisting of causal rules of $T \setminus T'$ and with the set of explainable symbols E''. Then

$$\begin{aligned} T &= T^*(E) \wedge \forall e[T^*(e) \rightarrow e = E] \\ &= T'^*(E) \wedge T''^*(E) \wedge \forall e'e''[T^*(e', e'') \rightarrow (e' = E' \wedge e'' = E'')]. \end{aligned}$$

The last conjunctive term entails

$$\forall e'[T^*(e', E'') \rightarrow (e' = E' \wedge E'' = E'')],$$

which is equivalent to

$$\forall e'[T'^*(e', E'') \wedge T''^*(e', E'') \rightarrow e' = E']. \tag{11.3}$$

Since the causal rules from T'' do not contain symbols from E' in the head, (11.3) can be written as

$$\forall e'[T'^*(e', E'') \wedge T''^*(E) \rightarrow e' = E'].$$

Thus T entails this formula. Since $T'^*(E)$ and $T''^*(E)$ are both present in T as conjunctive terms, it follows that T entails

$$T'^*(E) \wedge \forall e'[T'^*(e', E'') \rightarrow e' = E'].$$

This formula is exactly T'. ■

11.3.3 Proof of Proposition 1

Proposition 1 *Assume that D is a single-module MAD action description. For any sort name s declared in D, the causal theory D_m entails*

$$\forall x \left(s(x) \leftrightarrow \bigvee_{o : o \in |s|} x = o \right).$$

Proof It is easy to see that the rules of D_m containing sort names in the head are:

- $\neg s(x) \Leftarrow \neg s(x)$ for each sort name s,
- $SORT_o(o) \Leftarrow \top$ for each object name o,
- $s(x) \rightarrow s'(x) \Leftarrow \top$ for each element (s, s') of the set $\mathcal{E}$ of edges in the sort graph of D.

Let D_m^1 be the causal theory consisting of these rules, such that its set $\mathbf{s}$ of explainable symbols consists of all sort names in D. It is clear that no explainable symbols of D_m^1 occur in the head of any other rules in D_m. By Lemma 5, $D_m \models D_m^1$.

The causal theory D_m^1 satisfies the condition of Lemma 3, so that D_m^1 is equivalent to the parallel circumscription $CIRC[F(\mathbf{s}); \mathbf{s}]$, where $F(\mathbf{s})$ is

$$\bigwedge_o SORT_o(o) \ \wedge \bigwedge_{s,s' \,:\, (s,s') \in \mathcal{E}} \forall x \ (s(x) \rightarrow s'(x))$$

(the first conjunction extends over all object names o declared in D). $F(\mathbf{s})$ can be

equivalently rewritten as the universal closure of the formula

$$\bigwedge_o (\top \rightarrow SORT_o(o)) \ \wedge \bigwedge_{s,s' \,:\, (s,s') \in \mathcal{E}} (s(x) \rightarrow s'(x)),$$

which is definite in $\mathbf{s}$. By Lemma 2, $CIRC[F(\mathbf{s}); \mathbf{s}] \models \mathbf{E} = \mathbf{s}$, where $\mathbf{E}$ is the tuple of predicate expressions $\lambda \mathbf{x} \forall \mathbf{e_s}(F(\mathbf{e_s}) \rightarrow s(\mathbf{x}))$ for all sort names s (here $\mathbf{e_s}$ is a tuple of predicate variables e_s for each sort name s).

Since D^1_m is equivalent to $CIRC[F(\mathbf{s}); \mathbf{s}]$, $D^1_m \models \mathbf{E} = \mathbf{s}$, that is,

$$D^1_m \models \bigwedge_s \forall x \{ \, \forall \mathbf{e_s}[F(\mathbf{e_s}) \rightarrow e_s(x)] \leftrightarrow s(x) \, \}.$$

Therefore, to prove Proposition 1, it is sufficient to prove that, for any sort name s in D, the formula

$$\forall x \left[\forall \mathbf{e_s} \, (F(\mathbf{e_s}) \rightarrow e_s(x)) \leftrightarrow \bigvee_{o : o \in |s|} x = o \right] \tag{11.4}$$

is logically valid.

(i) Left to right: assume

$$\forall \mathbf{e_s} \, (F(\mathbf{e_s}) \rightarrow e_s(x)). \tag{11.5}$$

According to the definition of $F(\mathbf{s})$, $F(\mathbf{e_s})$ is

$$\bigwedge_o e_{SORT_o}(o) \ \wedge \bigwedge_{(s,s') \in \mathcal{E}} \forall x \ (e_s(x) \rightarrow e_{s'}(x)). \tag{11.6}$$

Then we can rewrite (11.5) as

$$\forall \mathbf{e_s} \left\{ \left[\bigwedge_{o} e_{SORT_o}(o) \ \wedge \bigwedge_{(s,s') \in \mathcal{E}} \forall x \ (e_s(x) \rightarrow e_{s'}(x)) \right] \rightarrow e_s(x) \right\}.$$

By substituting $\lambda x(\bigvee_{o \in |s|} \ x = o)$ for every e_s, we conclude

$$\left[\bigwedge_{o} \bigvee_{o' \in |SORT_o|} o = o' \ \wedge \bigwedge_{(s,s') \in \mathcal{E}} \forall x \left(\bigvee_{o \in |s|} x = o \ \rightarrow \bigvee_{o \in |s'|} x = o \right) \right] \rightarrow \bigvee_{o \in |s|} \ x = o. \tag{11.7}$$

The first conjunctive term in the antecedent of (11.7)

$$\bigwedge_{o} \bigvee_{o' \in |SORT_o|} o = o'$$

is logically valid because one of the disjunctive terms $o = o'$ is $o = o$ for any object name o.

The second conjunctive term in the antecedent of (11.7)

$$\bigwedge_{(s,s') \in \mathcal{E}} \forall x \left(\bigvee_{o \in |s|} x = o \ \rightarrow \bigvee_{o \in |s'|} x = o \right)$$

is also logically valid: it is equivalent to

$$\bigwedge_{(s,s') \in \mathcal{E}} \forall x \left[\bigwedge_{o \in |s|} \left(x = o \ \rightarrow \bigvee_{o \in |s'|} x = o \right) \right],$$

which is equivalent to

$$\bigwedge_{(s,s') \in \mathcal{E}} \bigwedge_{o \in |s|} \bigvee_{o' \in |s'|} o = o'.$$

By the definition of $|s|$, for every $o \in |s|$, s is reachable from $SORT_o$ in the sort

graph of D. If $(s, s') \in \mathcal{E}$, then s' is reachable from $SORT_o$ also. Thus $o \in |s'|$, and one of the disjunctive terms $o = o'$ is $o = o$.

Therefore, the antecedent of (11.7) is logically valid, and we conclude the consequent

$$\bigvee_{o:o\in|s|} \; x = o,$$

which is the right-hand side of (11.4).

(ii) Right to left: it is sufficient to prove that, for any o, s such that $o \in |s|$,

$$\forall x[\, x = o \rightarrow \forall \mathbf{e_s}(F(\mathbf{e_s}) \rightarrow e_s(x))\,]$$

is logically valid. This formula is equivalent to

$$\forall \mathbf{e_s}(F(\mathbf{e_s}) \rightarrow e_s(o)). \tag{11.8}$$

Since $o \in |s|$, s is reachable from $SORT_o$ in the sort graph of D. So there exists a path from $SORT_o$ to s in this graph. We will prove (11.8) by induction on the length l of that path.

Base case ($l = 0$): $s = SORT_o$. The consequent $e_s(o)$ is one of the conjunctive terms of the antecedent (11.6).

Induction step: Assume that there exists a path of length $l + 1$ from $SORT_o$ to s, then there exists a path of length l from $SORT_o$ to some sort s' such that $(s', s) \in \mathcal{E}$. By induction hypothesis, the formula

$$\forall \mathbf{e_s}(F(\mathbf{e_s}) \rightarrow e_{s'}(o))$$

is logically valid. On the other hand, the formula

$$\forall \mathbf{e_s}\{F(\mathbf{e_s}) \to \forall x[e_{s'}(x) \to e_s(x)]\}.$$

is logically valid because the consequent is one of the conjunctive terms of the antecedent (11.6). Thus the formula

$$\forall \mathbf{e_s}\{\ F(\mathbf{e_s}) \to [e_{s'}(o) \wedge \forall x(e_{s'}(x) \to e_s(x))]\ \} \tag{11.9}$$

is logically valid also, so is (11.8) since it is entailed by (11.9). ■

11.3.4 Proof of Proposition 2

Proposition 2 *For any transition I of D, the interpretations I^0 and I^1 are states of D.*

Proof It is clear that all causal rules of D_0 are causal rules of D_1, and that the explainable symbols of D_0 (sort names and $0:c$ for each statically determined fluent name c) are explainable symbols of D_1. Because the explainable symbols in D_0 don't occur in the heads of the other rules of D_1, by Lemma 5, $D_1 \models D_0$. Since I is a transition, I is a model of D_1, so I is a model of D_0 also. Since I^0 is I restricted to the signature σ^{D_0}, it follows that I^0 is a model of D_0, that is, a state.

Now let us prove that I^1 is a state. Let D_0^+ denote the causal theory obtained from D_0 by replacing every occurrence of "$0:$" with "$1:$" in the signature and in the causal rules. The set of explainable symbols of D_0^+ consists of every sort name and $1:c$ for every statically determined fluent name c. It is easy to see that it is a subset of the set of explainable symbols of D_1, and that the causal rules of D_0^+ is a subset of D_1. Because the explainable symbols of D_0^+ don't occur in the heads of

the other rules of D_1, by Lemma 5, $D_1 \models D_0^+$.

Recall that I^1 is defined by

$$
\begin{aligned}
&|I^1| = |I|,\\
&I^1[s] = I[s] \text{ for every sort name } s,\\
&I^1[o] = I[o] \text{ for every object name } o,\\
&I^1[0:c] = I[1:c] \text{ for every fluent name } c,\\
&I^1[\mathit{None}] = I[\mathit{None}].
\end{aligned}
$$

Let $(I^1)^+$ be the interpretation of the signature $\sigma^{D_0^+}$ such that

$$
\begin{aligned}
&|(I^1)^+| = |I|,\\
&(I^1)^+[s] = I[s] \text{ for every sort name } s,\\
&(I^1)^+[o] = I[o] \text{ for every object name } o,\\
&(I^1)^+[1:c] = I[1:c] \text{ for every fluent name } c,\\
&(I^1)^+[\mathit{None}] = I[\mathit{None}].
\end{aligned}
$$

It is easy to see that

$$(I^1)^+ \models D_0^+ \text{ iff } I^1 \models D_0. \tag{11.10}$$

On the other hand, $(I^1)^+$ is I restricted to $\sigma^{D_0^+}$.

Since I is a model of D_1, I is a model of D_0^+, which means that $(I^1)^+$ is a model of D_0^+. By (11.10), I^1 is a model of D_0. ■

11.3.5 Proof of Proposition 3

Proposition 3 *For any positive integer m and any interpretation I of σ^{D_m}, I is a model of D_m iff every $I^{(i)}$ $(0 \leq i < m)$ is a transition of D.*

Proof Proof by induction on m.

Base case ($m = 1$): it is trivial since $I^{(0)} = I$.

Induction step: assuming that the assertion of Proposition 3 holds for a positive integer m, we will prove that it holds for $m + 1$ also.

(i) Assume that an interpretation I of $\sigma^{D_{m+1}}$ is a model of D_{m+1}. It is clear that every causal rule of D_m is a causal rule of D_{m+1}, and every explainable symbol of is D_m is an explainable symbol of D_{m+1}. Since the explainable symbols of D_m do not occur in the heads of any other rules of D_{m+1}, by Lemma 5, $D_{m+1} \models D_m$. Thus I restricted to σ^{D_m} is a model of D_m. By the induction hypothesis, it follows that $I^{(i)}$ is a transition of D for every i such that $0 \leq i < m$. It remains to prove that $I^{(m)}$ is a transition of D.

Let T be the causal theory defined as follows. It consists of the following causal rules of D_{m+1} (see Section 10.2.2):

- rules (i), (ii), (iii), (iv) and (xi);
- rules (vi), (vii), and (viii) with time stamp $i = m$;
- rules (vi), (vii), and (viii) with time stamp $i = m + 1$;
- rules (v), (ix), and (x) with time stamp $j = m$.

Its set of explainable symbols includes every sort name, $m : c$ for each action name c, $m : c$ and $(m + 1) : c$ for each statically determined fluent name c, and $(m + 1) : c$ for each simple fluent name c. Its set of non-explainable symbols includes every object name, the object constant *None*, and $m : c$ for each simple fluent name c. According to the restrictions in the syntax of MAD (Section 8.2.3), every rule of D_{m+1} containing explainable symbols of T in the head belongs to T. By Lemma 5, $D_{m+1} \models T$. Thus I restricted to σ^T is a model of T.

On the other hand, T is exactly the causal theory obtained from D_1 by increasing every time stamp by m in the causal rules and in the signature (replacing every occurrence of "i :" with "$(i + m)$:"), and I restricted to σ^T is exactly the interpretation obtained from $I^{(m)}$ by increasing every time stamp by m. Therefore $I^{(m)}$ is a model of D_1, that is, a transition of D.

(ii) Assume that for an interpretation I of $\sigma^{D_{m+1}}$, $I^{(i)}$ is a transition of D for every i such that $0 \leq i < m + 1$. By the induction hypothesis, I restricted to σ^{D_m} is model of D_m.

Let D_1' be the causal theory defined as follows. It consists of the following causal rules of D_1 (see Section 10.2.2):

- rules (vi), (vii), and (viii) with time stamp $i = 1$;
- rules (v), (ix), and (x) with time stamp $j = 0$.

Its signature includes the same symbols in the signature of D_1, and its set of explainable symbols includes $0 : c$ for each action name c, and $1 : c$ for each fluent name c. By Lemma 5, $D_1 \models D_1'$. Since $I^{(m)}$ is a transition of D, $I^{(m)}$ is model of D_1' also. Let T' be the causal theory obtained from D_1' by increasing every time stamp by m in the causal rules and in the signature, and let I' be the interpretation of $\sigma^{T'}$ obtained from $I^{(m)}$ by increasing every time stamp by m. It follows that I' is model of T'.

On the other hand, I' is exactly I restricted to the signature of T'. Since I restricted to σ^{D_m} is model of D_m, it follows that I is model of $D_m \wedge T'$. Furthermore, it is easy to see that $D_{m+1} = D_m \cup T'$, and that T' and D_m are disjoint. By Lemma 4, it follows that I is model of D_{m+1}. ■

Chapter 12

Relationship Between the Two Semantics

In Sections 7.2 and 10.2, we presented two approaches to defining the semantics of modular action description languages. In the semantics of mini-MAD, grounding was used to turn every action description containing a single module into a set of causal laws in $\mathcal{C}+$. In the semantics of MAD, every action description containing a single module was translated into first-order causal logic. To relate these two views to each other, we will show that the second approach, when restricted to mini-MAD, is equivalent to the first, under the assumption that the universe of every sort is non-empty. (This assumption corresponds to the requirement, in the semantics of $\mathcal{C}+$, that the domain of every constant be a non-empty set.) In this chapter, D is an arbitrary mini-MAD action description with this property. Propositions 4 and 5 below show how models of $(cplus(D))_m$ in the sense of propositional causal logic (Section 3.1) and models of $(\delta D)_m$ in the sense of first-order causal logic (Section 9.1) can be characterized in terms of each other.

12.1 From First-order to Propositional Causal Logic

In the construction of interpretations of $(cplus(D))_m$, we refer to the following two conditions on an interpretation I of $\sigma^{(\delta D)_m}$:

(a) $I \models o_1 \neq o_2$ for any pair of distinct object names o_1, o_2, and $I \models o \neq None$ for any object name o;

(b) $I \models \forall \mathbf{x}\,[SORT_{c(1)}(x_1) \wedge \cdots \wedge SORT_{c(n)}(x_n) \rightarrow \bigvee_{o \in U_D(SORT_c)} i : c(\mathbf{x}) = o]$, for any non-Boolean fluent name c of arity n and any $i \in \{0, \ldots, m\}$, where $\mathbf{x}$ stands for $x_1, \ldots, x_n$.

For any interpretation I of $\sigma^{(\delta D)_m}$ satisfying these conditions, by I^{prop} we denote the interpretation (in the sense of propositional causal logic) of the signature $\sigma^{(cplus(D))_m}$ such that

- for each Boolean constant $i : c(o_1, \ldots, o_n)$,

$$I^{prop}[i : c(o_1, \ldots, o_n)] = I[i : c](I[o_1], \ldots, I[o_n]);$$

- for each non-Boolean fluent constant $i : c(o_1, \ldots, o_n)$, $I^{prop}[i : c(o_1, \ldots, o_n)]$ is the object o of $SORT_c$ such that $I[i : c](I[o_1], \ldots, I[o_n]) = I[o]$. (Conditions (a) and (b) guarantee the uniqueness and existence of such o.)

Any model I of $(\delta D)_m$ satisfies conditions (a) and (b) above because $(\delta D)_m$ contains the causal rules

$$o_1 \neq o_2 \Leftarrow \top$$

for all pairs of distinct object names o_1, o_2, and

$$o \neq None \Leftarrow \top$$

for all object names o, and

$$i{:}\,c(x_1,\dots,x_n) \neq y \;\Leftarrow\; \neg SORT_c(y) \wedge SORT_{c(1)}(x_1) \wedge \cdots \wedge SORT_{c(n)}(x_n)$$

for all non-Boolean fluent names c.

Proposition 4 *For any model I of $(\delta D)_m$ in the sense of first-order causal logic, I^{prop} is a model of $(cplus(D))_m$ in the sense of propositional causal logic.*

12.2 From Propositional to First-order Causal Logic

For any interpretation J of $\sigma^{(cplus(D))_m}$, by J^{fo} we denote the interpretation (in the sense of first-order logic) of the signature $\sigma^{(\delta D)_m}$ such that

- $|J^{fo}|$ consists of all object names and the symbol *None*,
- $J^{fo}[s](x) = \begin{cases} \mathbf{true} & \text{if } x \in U_D(s), \\ \mathbf{false} & \text{otherwise,} \end{cases}$
  for every sort name s,
- $J^{fo}[o] = o$ for every object name o,
- $J^{fo}[None] = None$,
- $J^{fo}[i:c](x_1,\dots,x_n)$
  $= \begin{cases} J[i:c(x_1,\dots,x_n)] & \text{if } x_k \in U_D(SORT_{c(k)}) \text{ for all } k, \\ \mathbf{f} & \text{otherwise,} \end{cases}$
  for every action name or Boolean fluent name c,
- $J^{fo}[i:c](x_1,\dots,x_n)$

$$= \begin{cases} J[i : c(x_1, \ldots, x_n)] & \text{if } x_k \in U_D(SORT_{c(k)}) \text{ for all } k, \\ None & \text{otherwise,} \end{cases}$$

for every non-Boolean fluent name c.

Note that (J^{fo}) satisfies conditions (a) and (b) from Section 12.1, and $(J^{fo})^{prop} = J$.

Proposition 5 *For any model J of $(cplus(D))_m$ in the sense of propositional causal logic, J^{fo} is a model of $(\delta D)_m$ in the sense of first-order causal logic.*

12.3 Proofs

12.3.1 Lemmas

The definition of $Cond(RC)$ in Section 10.1, restricted to constant renaming clauses of mini-MAD, is somewhat different from the definition of $Cond(RC)$ in Section 7.1. In the proofs of Propositions 4 and 5, this difference is essential, and we will write $Cond_{\text{mini}}$ when we refer to $Cond$ as defined in Section 7.1. The symbols ν_{mini} and δ_{mini} are understood in a similar way.

Lemma 6 describes the relationship between the functions δ and δ_{mini} in application to mini-MAD action descriptions.

Lemma 6 *For any positive integer m, the first-order causal theory $(\delta_{\text{mini}} D)_m$ is equivalent to $(\delta D)_m$.*

Proof Take any Boolean constant renaming clause (7.2) in D. It is clear that

$$Cond(RC) = Cond_{\text{mini}}(RC) \wedge \bigwedge_i SORT_{arg_i}(x_i), \tag{12.1}$$

where the big conjunction extends over all i such that arg_i is a variable. According to the sort matching conditions (Section 6.14), $SORT_{arg_i} = SORT_{c(i)}$. For any module

M from D, x_i is declared in the module $\nu(RC, M)$ to be of $SORT_{c(i)}$, so that (12.1) can be written as

$$Cond(RC) = Cond_{\text{mini}}(RC) \wedge \bigwedge_i SORT_{x_i}(x_i).$$

Consequently, module $\nu(RC, M)$ can be obtained from module $\nu_{\text{mini}}(RC, M)$ by replacing $Cond_{\text{mini}}(RC)$ with $Cond_{\text{mini}}(RC) \wedge \bigwedge_i SORT_{x_i}(x_i)$ in the bodies of some axioms of the forms

$$c(x_1, \ldots, x_n) \leftrightarrow rhs(x\_var) \textbf{ if } Cond_{\text{mini}}(RC)$$

and

$$\neg c(x_1, \ldots, x_n) \textbf{ if } \neg Cond_{\text{mini}}(RC).$$

It follows that δD can be obtained from $\delta_{\text{mini}} D$ by replacing G with $G \wedge H$ in the bodies of some axioms

$$F \textbf{ if } G \tag{12.2}$$

$$F \textbf{ if } \neg G, \tag{12.3}$$

where H is the conjunction of the formulas $SORT_x(x)$ for some variables x occurring in F. The causal rules of $(\delta_{\text{mini}} D)_m$ corresponding to (12.2) and (12.3) are

$$i{:}F \Leftarrow i{:}G \wedge \bigwedge_x SORT_x(x) \tag{12.4}$$

$$i{:}F \Leftarrow i{:}\neg G \wedge \bigwedge_x SORT_x(x) \tag{12.5}$$

where the conjunction extends over all variables x occurring in F or G. The causal

rules of $(\delta D)_m$ corresponding to (12.2) and (12.3) are

$$i{:}F \Leftarrow i{:}G \wedge H \wedge \bigwedge_x SORT_x(x) \tag{12.6}$$

$$i{:}F \Leftarrow \neg((i{:}G) \wedge H) \wedge \bigwedge_x SORT_x(x) \tag{12.7}$$

The body of (12.4) is equivalent to the body of (12.6), and the body of (12.5) is equivalent to the body of (12.7), because each conjunctive term of H is a conjunctive term of $\bigwedge_x SORT_x(x)$.

Therefore, $(\delta_{\text{mini}} D)_m$ and $(\delta D)_m$ are equivalent to each other. ■

In the signature $(\delta D)_m$ (Section 10.2.1), the symbol $0{:}c$ is non-explainable when c is a simple fluent name, and in mini-MAD all fluents are simple. Lemma 7 below shows that $(\delta D)_m$ is equivalent to a first-order causal theory obtained from $(\delta D)_m$ by making $0{:}c$ explainable and adding some causal rules. It is similar to Proposition 2 in [Lifschitz, 1997].

Lemma 7 *For any positive integer m, let T denote the corresponding first-order causal theory $(\delta D)_m$. Let T_1 be the first-order causal theory obtained from T by making $0{:}c$ explainable for some (possibly all) fluent names c and adding, for each of these fluent names, the causal rules*

$$\begin{aligned} 0{:}c(x_1,\dots,x_n) &\Leftarrow 0{:}c(x_1,\dots,x_n) \wedge SORT_{c(1)}(x_1) \wedge \cdots \wedge SORT_{c(n)}(x_n), \\ \neg\, 0{:}c(x_1,\dots,x_n) &\Leftarrow \neg\, 0{:}c(x_1,\dots,x_n) \wedge SORT_{c(1)}(x_1) \wedge \cdots \wedge SORT_{c(n)}(x_n), \end{aligned}$$

if c is a Boolean fluent name of arity n, and the causal rule

$$\begin{aligned} 0{:}c(x_1,\dots,x_n) = y \Leftarrow\ & 0{:}c(x_1,\dots,x_n) = y \\ & \wedge SORT_c(y) \wedge SORT_{c(1)}(x_1) \wedge \cdots \wedge SORT_{c(n)}(x_n), \end{aligned}$$

if c is a non-Boolean fluent name of arity n. Then T is equivalent to T_1.

Proof Clearly it is sufficient to prove this lemma when $0:c$ is made explainable for only one fluent name c.

According to the semantics of first-order causal theories (Section 9.1), T_1 is understood as the sentence

$$T_1^*(\mathbf{u}, v) \wedge \forall \mathbf{e_u} e_v (T_1^*(\mathbf{e_u}, e_v) \rightarrow (\mathbf{e_u} = \mathbf{u} \wedge e_v = v)), \tag{12.8}$$

where v denotes $0:c$, $\mathbf{u}$ denotes the list of all other explainable symbols in T_1 (that is, all explainable symbols in T), and e_v and $\mathbf{e_u}$ denote the corresponding variables.

The first conjunctive term in (12.8), $T_1^*(\mathbf{u}, v)$, is equivalent to $T^*(\mathbf{u}, v)$. On the other hand, the second conjunctive term in (12.8) can be written as

$$\forall \mathbf{e_u} e_v (T^*(\mathbf{e_u}, e_v) \wedge R(e_v) \rightarrow (\mathbf{e_u} = \mathbf{u} \wedge e_v = v)),$$

where $R(e_v)$ denote the conjunctive term in $T_1^*(\mathbf{e_u}, e_v)$ that corresponds to the additional causal rules as specified in the statement of Lemma 7.

It follows that (12.8) is equivalent to

$$T^*(\mathbf{u}, v) \wedge \forall \mathbf{e_u} e_v (T^*(\mathbf{e_u}, e_v) \wedge R(e_v) \rightarrow (\mathbf{e_u} = \mathbf{u} \wedge e_v = v)). \tag{12.9}$$

We consider two cases: when c is Boolean or non-Boolean.

(i) If c is Boolean, then $R(e_v)$ is

$$\forall \mathbf{x}(v(\mathbf{x}) \wedge SORT_{c()}(\mathbf{x}) \rightarrow e_v(\mathbf{x})) \wedge \forall \mathbf{x}(\neg v(\mathbf{x}) \wedge SORT_{c()}(\mathbf{x}) \rightarrow \neg e_v(\mathbf{x})),$$

where $\mathbf{x}$ stands for $x_1, \ldots, x_n$, and $SORT_{c()}(\mathbf{x})$ stands for $SORT_{c(1)}(x_1) \wedge \cdots \wedge$

$SORT_{c(n)}(x_n)$. This formula can be equivalently written as

$$\forall \mathbf{x}(SORT_{c()}(\mathbf{x}) \rightarrow (e_v(\mathbf{x}) \leftrightarrow v(\mathbf{x}))). \tag{12.10}$$

Because $(\delta D)_m$ contains the causal rule (see Section 10.2.2)

$$\neg 0 : c(x_1, \ldots, x_n) \;\Leftarrow\; \neg SORT_{c(1)}(x_1) \vee \cdots \vee \neg SORT_{c(n)}(x_n),$$

that is,

$$\neg v(\mathbf{x}) \;\Leftarrow\; \neg SORT_{c()}(\mathbf{x}),$$

the formula

$$T^*(\mathbf{u}, v) \rightarrow \forall \mathbf{x}(\neg SORT_{c()}(\mathbf{x}) \rightarrow \neg v(\mathbf{x}))$$

is logically valid, and so is

$$T^*(\mathbf{e_u}, e_v) \rightarrow \forall \mathbf{x}(\neg SORT_{c()}(\mathbf{x}) \rightarrow \neg e_v(\mathbf{x})).$$

Consequently, the formula

$$T^*(\mathbf{u}, v) \wedge T^*(\mathbf{e_u}, e_v) \rightarrow \forall \mathbf{x}(\neg SORT_{c()}(\mathbf{x}) \rightarrow (e_v(\mathbf{x}) \leftrightarrow v(\mathbf{x})))$$

is logically valid too. It follows that, in the presence of $T^*(\mathbf{u}, v)$ and $T^*(\mathbf{e_u}, e_v)$, (12.10) can be rewritten as

$$\forall \mathbf{x}(e_v(\mathbf{x}) \leftrightarrow v(\mathbf{x})),$$

that is, $e_v = v$. From this fact we can conclude that (12.9) is equivalent to

$$T^*(\mathbf{u}, v) \wedge \forall \mathbf{e_u} e_v (T^*(\mathbf{e_u}, e_v) \wedge (e_v = v) \rightarrow (\mathbf{e_u} = \mathbf{u} \wedge e_v = v)).$$

This formula is equivalent to

$$T^*(\mathbf{u}, v) \wedge \forall \mathbf{e_u}(T^*(\mathbf{e_u}, v) \rightarrow (\mathbf{e_u} = \mathbf{u})),$$

that is, to T.

(ii) If c is non-Boolean, $R(e_v)$ is

$$\forall \mathbf{x}y(v(\mathbf{x}) = y \wedge SORT_c(y) \wedge SORT_{c()}(\mathbf{x}) \rightarrow e_v(\mathbf{x}) = y). \tag{12.11}$$

Because $(\delta D)_m$ contains the causal rule (see Section 10.2.2)

$$0{:}\, c(\mathbf{x}) \neq y \;\Leftarrow\; \neg SORT_c(y) \wedge SORT_{c(1)}(x_1) \wedge \cdots \wedge SORT_{c(n)}(x_n),$$

that is,

$$v(\mathbf{x}) \neq y \;\Leftarrow\; \neg SORT_c(y) \wedge SORT_{c()}(\mathbf{x}),$$

the formulas

$$T^*(\mathbf{u}, v) \rightarrow \forall \mathbf{x}y(\neg SORT_c(y) \wedge SORT_{c()}(\mathbf{x}) \rightarrow v(\mathbf{x}) \neq y),$$
$$T^*(\mathbf{e_u}, e_v) \rightarrow \forall \mathbf{x}y(\neg SORT_c(y) \wedge SORT_{c()}(\mathbf{x}) \rightarrow e_v(\mathbf{x}) \neq y)$$

are logically valid. Thus, in the presence of $T^*(\mathbf{u}, v)$ and $T^*(\mathbf{e_u}, e_v)$, (12.11) can be equivalently rewritten as

$$\forall \mathbf{x}(SORT_{c()}(\mathbf{x}) \rightarrow (e_v(\mathbf{x}) = v(\mathbf{x}))). \tag{12.12}$$

On the other hand, because $(\delta D)_m$ contains the causal rule (see Section 10.2.2)

$$0{:}\, c(\mathbf{x}) = None \;\Leftarrow\; \neg SORT_{c(1)}(x_1) \vee \cdots \vee \neg SORT_{c(n)}(x_n),$$

that is,

$$v(\mathbf{x}) = None \;\Leftarrow\; \neg SORT_{c()}(\mathbf{x}),$$

the formulas

$$T^*(\mathbf{u}, v) \to \forall \mathbf{x}(\neg SORT_{c()}(\mathbf{x}) \to v(\mathbf{x}) = None),$$
$$T^*(\mathbf{e_u}, e_v) \to \forall \mathbf{x}(\neg SORT_{c()}(\mathbf{x}) \to e_v(\mathbf{x}) = None)$$

are logically valid. Consequently,

$$T^*(\mathbf{u}, v) \wedge T^*(\mathbf{e_u}, e_v) \to \forall \mathbf{x}(\neg SORT_{c()}(\mathbf{x}) \to (e_v(\mathbf{x}) = v(\mathbf{x})))$$

is logically valid too. It follows that, in the presence of $T^*(\mathbf{u}, v)$ and $T^*(\mathbf{e_u}, e_v)$, (12.12) is equivalent to

$$\forall \mathbf{x}(e_v(\mathbf{x}) = v(\mathbf{x})),$$

that is, $e_v = v$. Similarly to the case when c is Boolean, from this fact we can conclude that (12.9) is equivalent to T.

Therefore, T_1 is equivalent to T. ■

Lemma 8 below generalizes Proposition 5 in [Lifschitz, 1997] from propositional constants to multi-valued constants.

Recall that a multi-valued signature is a set of constants, with a domain assigned to each of them (Section 3.1). Any such signature σ can be extended to a signature $fo(\sigma)$ in the sense of first-order causal logic as follows: non-Boolean constants from σ are explainable object constants; elements of their domains are non-explainable object constants; Boolean constants from σ are explainable propositional constants. Any formula of a multi-valued signature σ can be viewed as a quantifier-free formula of $fo(\sigma)$ in the sense of first-order logic if we identify $c = \mathbf{true}$ with c

and $c = \mathbf{false}$ with $\neg c$ for all Boolean constants c.

Any interpretation I of σ in the sense of Section 3.1 can be extended to an interpretation of $fo(\sigma)$ in the sense of first-order logic as follows:

- $|I|$ is the union of the domains of all non-Boolean constants of σ,
- $I[v] = v$ for each non-explainable object constant v.

An interpretation of $fo(\sigma)$ is called *regular* if it satisfies the two conditions above. We will identify an interpretation of σ in the sense of Section 3.1 with the corresponding regular interpretation of $fo(\sigma)$ in the sense of first-order logic.

Let T be a finite multi-valued propositional causal theory of a signature σ. By $fo(T)$ we denote the first-order causal theory whose signature is $fo(\sigma)$ and whose causal rules are those of T and the causal rules

$$\bigvee_{v \in Dom(c)} c = v \Leftarrow \top$$

for all non-Boolean constants c from σ.

Lemma 8 *Let T be a finite multi-valued propositional causal theory of a signature σ, and let I be a regular interpretation of $fo(\sigma)$. Then I is a model of $fo(T)$ iff I is an interpretation of σ and is a model of T.*

Proof Let

$$F_i(\mathbf{p}, \mathbf{c}) \Leftarrow G_i(\mathbf{p}, \mathbf{c}) \qquad (i = 1, \dots)$$

be the rules of T, where $\mathbf{p}$ is the list of all Boolean constants p_j and $\mathbf{c}$ is the list of all non-Boolean constants c_j of σ. Let T_1 be the corresponding first-order causal

theory $fo(T)$. Then T_1 is understood as the sentence

$$\forall \mathbf{e_p e_c}(T_1^*(\mathbf{e_p}, \mathbf{e_c}) \leftrightarrow (\mathbf{e_p} = \mathbf{p} \wedge \mathbf{e_c} = \mathbf{c})),$$

where $T_1^*(\mathbf{e_p}, \mathbf{e_c})$ stands for

$$\bigwedge_i (G_i(\mathbf{p}, \mathbf{c}) \rightarrow F_i(\mathbf{e_p}, \mathbf{e_c})) \wedge \bigwedge_{c_j \in \mathbf{c}} \left(\bigvee_{v \in Dom(c_j)} e_{c_j} = v \right).$$

Take a regular interpretation I, and let $\hat{I}(\mathbf{p}, \mathbf{c})$ be the formula

$$\bigwedge_{j: I \models p_j} p_j \ \wedge \bigwedge_{j: I \not\models p_j} \neg p_j \ \wedge \bigwedge_j (c_j = I[c_j]).$$

It is clear that I is the only regular interpretation of $fo(\sigma)$ that satisfies $\hat{I}(\mathbf{p}, \mathbf{c})$. Consequently I is a model of T_1 iff the formula

$$\hat{I}(\mathbf{p}, \mathbf{c}) \wedge \forall \mathbf{e_p e_c}(T_1^*(\mathbf{e_p}, \mathbf{e_c}) \leftrightarrow (\mathbf{e_p} = \mathbf{p} \wedge \mathbf{e_c} = \mathbf{c}))$$

is satisfied by some regular interpretation of $fo(\sigma)$. Consider the reduct T_1^I of T relative to I (Section 3.1), we will write the conjunction of the formulas from T_1^I as $T_1^I(\mathbf{p}, \mathbf{c})$. In the presence of $\hat{I}(\mathbf{p}, \mathbf{c})$, $T_1^*(\mathbf{e_p}, \mathbf{e_c})$ can be replaced by $T_1^I(\mathbf{e_p}, \mathbf{e_c})$, and $\mathbf{e_p} = \mathbf{p} \wedge \mathbf{e_c} = \mathbf{c}$ can be replaced by $\hat{I}(\mathbf{e_p}, \mathbf{e_c})$. Consequently, I is a model of T_1 iff the formula

$$\hat{I}(\mathbf{p}, \mathbf{c}) \wedge \forall \mathbf{e_p e_c}(T_1^I(\mathbf{e_p}, \mathbf{e_c}) \leftrightarrow \hat{I}(\mathbf{e_p}, \mathbf{e_c}))$$

is satisfied by some regular interpretation of $fo(\sigma)$. Since I is the only regular interpretation satisfying the first conjunctive term $\hat{I}(\mathbf{p}, \mathbf{c})$, it follows that I is a

model of T_1 iff I satisfies the second conjunctive term

$$\forall \mathbf{e_p e_c}(T_1^I(\mathbf{e_p}, \mathbf{e_c}) \leftrightarrow \hat{I}(\mathbf{e_p}, \mathbf{e_c})). \tag{12.13}$$

But (12.13) contains no nonlogical constants except for the constants v such that $J[v] = v$ for every regular interpretation J. Consequently if I satisfies (12.13) then so does every regular interpretation. Thus I is a model of T_1 iff (12.13) is satisfied by all regular interpretations, and consequently iff the formula

$$T_1^I(\mathbf{p}, \mathbf{c}) \leftrightarrow \hat{I}(\mathbf{p}, \mathbf{c}) \tag{12.14}$$

is satisfied by all regular interpretations of $fo(\sigma)$.

The condition "every regular interpretation satisfies the right-to-left implication in (12.14)" means that I satisfies $T_1^I(\mathbf{p}, \mathbf{c})$. Since $T_1^I(\mathbf{p}, \mathbf{c})$ is the conjunction of $T^I(\mathbf{p}, \mathbf{c})$ and $\bigwedge_{c_j \in \mathbf{c}} \left(\bigvee_{v \in Dom(c_j)} c_j = v \right)$, this can be expressed by saying that I satisfies $T^I(\mathbf{p}, \mathbf{c})$ and is an interpretation of σ.

The condition "every regular interpretation satisfies the left-to-right implication in (12.14)" means that every interpretation of σ satisfying $T^I(\mathbf{p}, \mathbf{c})$ equals I.

Consequently, I is a model of T_1 iff I is an interpretation of σ and a model of T. ■

12.3.2 Proof of Propositions 4 and 5

Proposition 4 (Section 12.1) and Proposition 5 (Section 12.2) easily follow from Lemma 9 below. In the statement of Lemma 9, we refer to conditions (a) and (b) in Section 12.1, and to the following condition (c) on an interpretation I of $\sigma^{(\delta D)_m}$:

(c) $I[s](x) = \begin{cases} \textbf{true} & \text{if } x = I[o] \text{ for some } o \in U_D(s), \\ \textbf{false} & \text{otherwise.} \end{cases}$

Lemma 9 *For any interpretation I of $\sigma^{(\delta D)_m}$ such that I satisfies conditions (a), (b) and (c), I is a model of $(\delta D)_m$ in the sense of first-order causal logic iff I^{prop} is a model of $(cplus(D))_m$ in the sense of propositional causal logic.*

To derive Proposition 4, observe that any model of $(\delta D)_m$ satisfies conditions (a), (b) and (c) (Section 12.1 and Proposition 1 from Section 11.1).

To derive Proposition 5, observe that for any model J of $(cplus(D))_m$, J^{fo} satisfies conditions (a), (b) and (c), and $(J^{fo})^{prop} = J$.

Proof of Lemma 9 Let T denote the causal theory $(\delta D)_m$, T_1 denote $(\delta_{\text{mini}} D)_m$, and T_2 denote the causal theory obtained from T_1 by making the symbols $0 : c$ explainable for each fluent name c in D and adding the causal rules shown in the statement of Lemma 7. By Lemmas 6 and 7, T is equivalent to T_1 and T_1 is equivalent to T_2. Thus T is equivalent to T_2. Divide T_2 into two parts T_{2s} and T_{2c}, as follows. T_{2s} consists of rules (i), (ii), (iii), and (iv) described in Section 10.2.2, with only sort names treated as explainable. T_{2c} consists of all other rules of T_2, with only the symbols $i : c$ treated as explainable. It is clear that T_{2s} and T_{2c} are disjoint (there are no rules of form (xi) in mini-MAD since no sort inclusion expressions are allowed). By Lemma 4 (Section 11.3.2), T_2 is equivalent to $T_{2s} \wedge T_{2c}$. Since I satisfies conditions (a) and (c), I satisfies T_{2s}. Therefore, I satisfies T_2 iff I satisfies T_{2c}.

Recall that causal theory T_{2c} stands for

$$\forall e(T^*_{2c}(e) \leftrightarrow e = E),$$

which is equivalent to

$$T^*_{2c}(E) \wedge \forall e(T^*_{2c}(e) \to e = E), \tag{12.15}$$

where E is the list of all explainable symbols in T_{2c} (that is, $i\!:\!c$ for all action names and fluent names c), and e is a list of variables corresponding to E.

Let u denote an arbitrary symbol in E. Let e_u denote an arbitrary function that maps elements of $|I|^{k_u}$ to truth values if u is predicate constant, and to elements of $|I|$ if u is function constant, where k_u is the arity of u. Add (a symbol for) each e_u to the signature as a predicate constant of the same arity as u if u is a predicate constant, and as a function constant of the same arity as u if u is a function constant. Extend the definition of I to e_u so that $I[e_u](\mathbf{x}) = e_u(\mathbf{x})$. Then I satisfies (12.15) iff I satisfies $T^*_{2c}(E)$ and I satisfies

$$T^*_{2c}(e_{u_1}, \ldots, e_{u_n}) \to ((e_{u_1} = u_1) \wedge \cdots \wedge (e_{u_n} = u_n)) \tag{12.16}$$

for all tuples $(e_{u_1}, \ldots, e_{u_n})$, where $u_1, \ldots, u_n$ are all symbols in E.

About a predicate constant e_u we say that it is *good* if

$$e_u(I[x_1], \ldots, I[x_{k_u}]) = \mathbf{true}$$

whenever $x_i \in U_D(SORT_{u(i)})$ for all i. About a function constant e_u we say that it is *good* if

- $e_u(I[x_1], \ldots, I[x_{k_u}]) = I[o]$ for some o in $U_D(SORT_u)$ whenever, for all i, $x_i \in U_D(SORT_{u(i)})$, and
- $e_u(I[x_1], \ldots, I[x_{k_u}]) = I[None]$ if some $x_i \notin U_D(SORT_{u(i)})$.

Since T_{2c} contains the rules

$$\neg\, i : c(x_1, \ldots, x_n) \;\Leftarrow\; \neg SORT_{c(1)}(x_1) \vee \cdots \vee \neg SORT_{c(n)}(x_n) \tag{12.17}$$

for every Boolean fluent name c of arity n in D, and the rules

$$\begin{aligned} &i : c(x_1, \ldots, x_n) = None \;\Leftarrow\; \neg SORT_{c(1)}(x_1) \vee \cdots \vee \neg SORT_{c(n)}(x_n) \\ &i : c(x_1, \ldots, x_n) \neq y \;\Leftarrow\; \neg SORT_c(y) \wedge SORT_{c(1)}(x_1) \wedge \cdots \wedge \neg SORT_{c(n)}(x_n) \end{aligned} \tag{12.18}$$

for every non-Boolean fluent name c of arity n in D, $T^*_{2c}(e_{u_1}, \ldots, e_{u_n})$ contains conjunctive terms that are implications corresponding to these rules. Since I satisfies condition (c), it follows that, if any of these e_u's is not good, then I doesn't satisfy these terms. Consequently I doesn't satisfy the formula $T^*_{2c}(e_{u_1}, \ldots, e_{u_n})$. Thus it is enough to consider only good e_u's in (12.16).

Furthermore, $T^*_{2c}(e_{u_1}, \ldots, e_{u_n})$ is a conjunction of formulas

$$\forall \mathbf{x} F_l(e_{u_1}, \ldots, e_{u_n}, \mathbf{x}) \tag{12.19}$$

corresponding to the causal rules l of T_{2c}. I satisfies (12.19) iff I satisfies

$$F_l(e_{u_1}, \ldots, e_{u_n}, \mathbf{y}) \tag{12.20}$$

for all tuples $\mathbf{y}$ of elements of $|I|$. For every variable x from $\mathbf{x}$ and every $y \in |I|$, we say that y is a *good* value (for x) if $y \in U_D(SORT_x)$. Since I satisfies condition (c), it follows that $I[SORT_x](y) = \mathbf{true}$ iff y is good.

Since each e_u is good, and I satisfies condition (c), I also satisfies all conjunctive terms (12.19) of $T^*_{2c}(e_{u_1}, \ldots, e_{u_n})$ corresponding to the rules l of the forms (12.17)

and (12.18). Consequently these terms can be dropped from $T^*_{2c}(e_{u_1}, \ldots, e_{u_n})$.

Every causal rule of T_{2c} other than (12.17) and (12.18) contains the conjunctive term $SORT_x(x)$ in the body for every variable x occurring in that rule. Consequently (12.20) can be limited to the tuples $\mathbf{y}$ whose members are good values, and the conjunctive terms $SORT_x(y)$ can be dropped from the antecedents of the implications (12.20). We will denote the formula obtained in this way from (12.20) by $F'_l(e_{u_1}, \ldots, e_{u_n}, \mathbf{y})$.

In the presence of $T^*_{2c}(E)$, I satisfies the conjunctive terms $\forall \mathbf{x} F_l(E, \mathbf{x})$ of $T^*_{2c}(E)$ corresponding to rules l of the forms (12.17) and (12.18) since I satisfies condition (c). Thus $I[u]$ is good for each u of E. Similarly to the reasoning above, I satisfies $T^*_{2c}(E)$ iff it satisfies $\bigwedge_l F'_l(E, \mathbf{y})$ for all tuples $\mathbf{y}$ of good values, where the conjunction extends over all causal rules in T_{2c} except (12.17) and (12.18).

On the other hand, $e = E$ stands for a conjunction of formulas

$$\forall \mathbf{x}(e_u(\mathbf{x}) = u(\mathbf{x}))$$

corresponding to all members u of E. I satisfies such a conjunctive term iff I satisfies

$$e_u(\mathbf{y}) = u(\mathbf{y}) \tag{12.21}$$

for all tuples $\mathbf{y}$ of elements of $|I|$. Since each e_u is good and each $I[u]$ is good in the presence of $T^*_{2c}(E)$, it follows that (12.21) can be limited to the tuples $\mathbf{y}$ whose members are good values.

Therefore, I satisfies T_2 is iff I satisfies

$$\bigwedge_{l}\bigwedge_{\mathbf{y}} F'_l(E,\mathbf{y}) \wedge \bigwedge_{(e_{u_1},\ldots,e_{u_n})} \Bigg[\bigwedge_{l}\bigwedge_{\mathbf{y}} F'_l(e_{u_1},\ldots,e_{u_n},\mathbf{y}) \rightarrow \bigwedge_{i=1}^{n}\bigwedge_{\mathbf{y}}(e_{u_i}(\mathbf{y}) = u_i(\mathbf{y}))\Bigg], \tag{12.22}$$

where the conjunction over $(e_{u_1},\ldots,e_{u_n})$ extends over all good predicate or function constants, the conjunction over $\mathbf{y}$ extends over all tuples of good values, and the conjunction over l extends over all causal rules in T_{2c} except (12.17) and (12.18).

Let T_{cp} denote the multi-valued propositional causal theory $(cplus(D))_m$, and let T_{cp1} denote the first-order causal theory $fo(T_{cp})$ (see the discussion before Lemma 8 for the definition of fo and of extending I^{prop} to an interpretation of the signature of T_{cp1}). So I satisfies (12.22) iff I^{prop} satisfies

$$T^*_{cp1}(E') \wedge \bigwedge_{e'}(T^*_{cp1}(e') \rightarrow e' = E'), \tag{12.23}$$

where E' is the list of constants defined in the signature of T_{cp}, and the conjunction extends over all interpretations e' of E'.

Furthermore, I^{prop} satisfies (12.23) iff it satisfies

$$T^*_{cp1}(E') \wedge \forall e'(T^*_{cp1}(e') \rightarrow e' = E'), \tag{12.24}$$

where e' is a tuple of variables corresponding to E', that is, iff I^{prop} is a model of T_{cp1} in the sense of first-order causal logic. Since I^{prop} is defined as an interpretation of the signature of T_{cp}, it is regular. By Lemma 8, I^{prop} satisfies (12.24) iff I^{prop} is a model of T_{cp} in the sense of propositional causal logic. Therefore, any I that satisfies (a), (b) and (c) is a model of T iff I^{prop} is a model of T_{cp}. ■

Chapter 13

Related Work

13.1 Modularity in Knowledge Representation

Our work on developing MAD from language $\mathcal{C}+$ is one of several efforts directed towards enhancing existing knowledge representation formalisms, including action languages, by adding modularity.

An attempt to add modularity to a planning formalism is outlined in [Clark *et al.*, 1996], where STRIPS operators are formed from "components", similar to modules in this dissertation. STRIPS is less expressive than the language $\mathcal{C}+$ that we begin with, and the "composition" of components is not as flexible as "import" in MAD.

The applicability of the object-oriented paradigm to modeling dynamic domains was investigated by Gustafsson and Kvarnström [2004]. Their system is based on Temporal Action Logic [Doherty and Kvarnström, 2008]. That work deals with a single domain (Missionaries and Cannibals) with many variations, and their focus is on showing how the object-oriented paradigm can bring structure to formalizing a domain, making it more understandable and easier to extend. The modularity in

[Gustafsson and Kvarnström, 2004] comes from classes, associating a set of fluents and axioms with each object of that class (sort), which can then be used to create new subclasses. In contrast, modules in MAD are more general: they are essentially action descriptions, not focused on a particular sort, though it is possible, in principle, to make MAD modules which mirror the notion of a class. Moreover, MAD allows us to rename sorts and constants when a module is imported, which seems to have no counterpart in their framework of [Gustafsson and Kvarnström, 2004].

The object-oriented paradigm has been applied also to first-order logic [Amir, 1999].

Research on modular logic programming is described in [Bugliesi *et al.*, 1994]. It is not directly related to the area of reasoning about actions. The idea of adding modularity to answer set programming has been explored in [Ianni *et al.*, 2004] and [Calimeri *et al.*, 2004]. These authors introduce "templates" as generic subprograms with some predicates used to parameterize the program. Their examples are focused on "aggregates" in logic programming, rather than on commonsense reasoning or describing actions.

In [Baral *et al.*, 2006], the authors added modularity to answer set programming by using macros and "ensembles" (groups of macros). In a macro call, terms can be added, removed, or replaced by other terms. The goal of that research is to enable the creation of a library of knowledge modules in Answer Set Programming (ASP) languages, which is similar to the goal of the MAD project. According to that paper, a module in an ASP language contains a name, a list of parameters, and a list of rules. *Call-macro statements* can be written to use a module, possibly in many different ways. Call-macro statements specify how a module is used, in particular, how to substitute variables or predicates, how to add or remove arguments

to predicates, and how to add or remove negation-as-failure literals from the bodies of rules. This is similar to importing a module with renaming clauses in MAD. For instance, a module that describes transitive closure is given in [Baral *et al.*, 2006] as follows:

```
Module_name: Transitive_closure.
Parameters(Input: p(X,Y); Output: q(X,Y);). Types: Z = type X.
Body: q(X,Y) :- p(X,Y).
      q(X,Y) :- p(X,Z), q(Z,Y).
```

Then the macro call

CallMacro Transitive_closure(**Replace** : p **by** $parent, q$ **by** $ancestor, X$ **by** U, Y **by** V;)

will be expanded to

```
ancestor(U,V) :- parent(U,V).
ancestor(U,V) :- parent(U,Z), ancestor(Z,V).
```

The authors of the paper point out that their parameter matching mechanism is inspired by our work on MAD [Lifschitz and Ren, 2006].

Gelfond [2006] outlined a language $\mathcal{M}$ for defining knowledge modules and for assembling them into a coherent knowledge base in the logic programming language CR-Prolog [Balduccini and Gelfond, 2003, Balduccini, 2007]. It 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)." The enhanced formalization emphasizes the possibility of interruption of a sequence of intended actions by unexpected and unplanned events (in other words, unexpected stops in the middle of a journey).

Adding modularity to CR-Prolog is similar to adding modular structure to $\mathcal{C}+$ in [Lifschitz and Ren, 2006].

Recently, Gelfond and Inclezan [2009] designed another modular action language similar to MAD, called $\mathcal{ALM}$. It extends action language $\mathcal{AL}$ [Baral and Gelfond, 2000] by allowing definitions of new actions or fluents in terms of other, previously defined, actions or fluents. A *system description D* in $\mathcal{ALM}$ consists of declarations and structures. Sort names and inclusion relations, actions and fluent are declared in declarations part. Objects of every sort, "instances" of actions (actions whose arguments are specified with objects or variables), and values (and/or relations) of static fluents are specified in structures. An action can be declared "from scratch" by specifying its *attributes* (essentially arguments) and listing axioms about it, or it can be declared in terms of other actions. $\mathcal{ALM}$ has no counterpart for MAD sort renaming clauses, and a new action in $\mathcal{ALM}$ must have at least as many arguments as the one previously declared. Thus the mechanism of describing actions in terms of previously defined actions in $\mathcal{ALM}$ is somewhat less flexible than importing modules in MAD. In $\mathcal{ALM}$ all fluents are Boolean, and there are no variable declarations. Language $\mathcal{ALM}$ seems to be somewhat less expressive than MAD, but it is more closer to answer set programming languages. The authors are currently working on proving some mathematical properties of $\mathcal{ALM}$ and implementing a transition of its theories into logic programs.

13.2 A Library of General-Purpose Action Descriptions in MAD

Erdoğan [2008] built a database of action description modules using the action language MAD. The core library contains modules of two kinds: one describing

general properties of actions (such as the need for the actor and the theme of an action to be at the same place); the other describing some basic actions which are likely to be imported in many applications, such as *Move* and *Mount*.

The core library includes the following modules:

- ACTOR and THEME introduce the concepts of actor and theme of an action.
- ORDER introduces the transitive, irreflexive relation *Less* among objects.
- ASSIGN describes how the action *Assign* affects the fluent constant *Value*.
- MOVE imports module ASSIGN to describe the effect of the action *Move* on the fluent *Location*.
- MOUNT imports modules ASSIGN, ORDER and THEME to describe support relations between things, and how the action *Mount* affects them.
- TOWER imports module MOUNT to describe a special kind of support structures.
- TOP postulates that, when a thing is held up by another thing, both of them are usually at the same location.
- NOCONCURRENCY states the condition that no two actions can be executed concurrently.
- LOCAL postulates that in general all actors and themes must be at the same location when an action is executed.

Several action domains from the knowledge representation literature are formalized in [Erdoğan, 2008] using the library of basic action descriptions. The availability of the library led to more concise representations and also allows the author

numeric_symbol $MaxBlocks = 3$;

module BW_SIMPLE;

import TOWER;

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

import NOCONCURRENCY;

axioms
$Wide(Table)$;

Figure 13.1: Formalizing the Blocks World domain

to recognize structural similarities of seemingly quite domains. This fact illustrates some advantages of using MAD to describe actions.

For instance, the Blocks World domain and the Towers of Hanoi domain are described by importing modules TOWER and NOCONCURRENCY, because both domains describe "tower structures" and putting things on top of each other—blocks in the Blocks World, and discs in the Towers of Hanoi, see Figures 13.1 and 13.2 ([Erdoğan, 2008, Sections 8.1 and 8.2]).

numeric_symbol $NumDisks = 3$;

module TOWER_OF_HANOI;

import TOWER;

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

variables
i, j: $1..NumDisks$;

import NOCONCURRENCY;

axioms
constraint $Support(D_i) = D_j \rightarrow i < j$;

Figure 13.2: Formalizing the Towers of Hanoi domain

Chapter 14

Conclusion and Future Work

14.1 Conclusion

In this dissertation we designed MAD—a modular language for describing actions. The possibility to import a module allows us to describe actions in this language in terms of other actions. This is often more natural and convenient than describing every action independently.

We started by defining the syntax of a fragment of MAD, called mini-MAD. An action description in mini-MAD is a list of sort declarations and modules. The semantics of mini-MAD is defined in two steps. First, a function δ turns an arbitrary mini-MAD description into a single-module description by eliminating all import statements. Second, grounding is used to translate the result of the first step into $\mathcal{C}+$.

Then we described the syntax of full MAD. It includes a few additional constructs that are useful for knowledge representation purposes: sort inclusion expressions, constants as arguments of other constants, sort names as unary predicates, and quantifiers. The definition of δ can be easily extended to MAD, but extending

the grounding process to MAD is difficult. For this reason, we developed an alternative approach to the semantics of variables, which is based on first-order causal logic

We stated three theorems on properties of MAD. They show that, for any MAD action description D,

- in any model of $(\delta D)_m$, the extent of any sort S corresponds to the set of the object names that one would expect to belong to S in accordance with the object declarations and sort inclusions of D;
- any transition "starts" in a state and "ends" in a state;
- an interpretation of the signature of $(\delta D)_m$ is a model of $(\delta D)_m$ if and only if it "consists of m transitions."

Moreover, we proved that in application to mini-MAD action descriptions the two semantics are equivalent to each other.

To sum up, MAD is an action description language with powerful expressive possibilities and well-defined semantics. In Selim Erdoğan's dissertation, a dialect of this language has been implemented and used for representing several action domains in a concise and elegant way. We expect that research on MAD will contribute to solving the problem of generality in AI.

14.2 Future work

The syntax presented in Chapter 8 can be extended by several useful features. It would be useful, for instance, to allow the right-hand sides of non-Boolean constant renaming clauses to be more general than in the current version. It would be useful

also to allow renaming objects, in addition to renaming sorts and constants. Attributes of actions and defeasible causal laws, familiar from $\mathcal{C}+$ [Giunchiglia *et al.*, 2004]), need to be extended to MAD.

The implementation of MAD in [Erdoğan, 2008] operates by generating a CCalc input file and invoking CCalc for search. We would like to develop an implementation of a different kind—a procedure that translates from MAD into into an implemented language of answer set programming and performs search using an ASP solver. Such a procedure may handle equivalences in the heads of causal rules (Section 7.1.1) using a new approach described in [Lee *et al.*, 2009]. That paper shows that the familiar transformation converting definite causal theories into logic programs [McCain, 1997] can be extended to causal theories containing nondefinite rules of the form

$$p \leftrightarrow q \ \Leftarrow .$$

This extension turns this rule into a group of 4 logic programming rules if that rule is turned into a group of logic programming rules with strong negation:

$$\begin{aligned} p &\ \leftarrow\ q, \\ q &\ \leftarrow\ p, \\ \neg p &\ \leftarrow\ \neg q, \\ \neg q &\ \leftarrow\ \neg p. \end{aligned}$$

Future work in this direction will require proving the correctness of translations of this kind, and thus will have a significant theoretical component.

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.

[Amir, 1999] Eyal Amir. Object-oriented first-order logic.[1] *Electronic Transactions on Artificial Intelligence*, 4:63–84, 1999.

[Armando *et al.*, 2009] Alessandro Armando, Enrico Giuchiglia, and Serena Elisa Ponta. Formal specification and automatic analysis of business processes under authorization constraints: an action-based approach. In *Proceedings of the 6th International Conference on Trust, Privacy and Security in Digital Business (TrustBus'09)*, 2009.

[Artificial Intelligence, 1980] *Artificial Intelligence*, 13(1,2), 1980.

[1] `http://www.ep.liu.se/ea/cis/1999/042/` .

[Artikis *et al.*, 2009] ALexander Artikis, Marek Sergot, and Jeremy Pitt. Specifying norm-governed computatinoal societies. *ACM Transactions on Computational Logic*, 9(1), 2009.

[Balduccini and Gelfond, 2003] Marcello Balduccini and Michael Gelfond. Logic programs with consistency-restoring rules.[2] 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.

[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 *Workshop on Logic-Based Artificial Intelligence*, pages 257–279, 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.

[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.

[Bugliesi *et al.*, 1994] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

[2] `http://www.krlab.cs.ttu.edu/papers/ download/bg03.pdf .`

[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.

[Clark *et al.*, 1996] Peter Clark, Bruce Porter, and Don Batory. A compositional approach to representing planning operators.[3] Technical Report AI06-331, University of Texas at Austin, 1996.

[Clark, 1978] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[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 *Proc. 7th Int'l Conference on Logic Programming and Nonmonotonic Reasoning*, pages 74–86, 2004. Extended version with proofs[4].

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system DLV: Progress report, comparisons and benchmarks. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 406–417, 1998.

[Erdoğan and Lifschitz, 2005] Selim T. Erdoğan and Vladimir Lifschitz. Actions as

[3] `http://www.cs.utexas.edu/ftp/pub/AI-Lab/tech-reports/UT-AI-TR-06-331.pdf` .
[4] `http://www.cs.utexas.edu/users/otto/papers/adct.ps`

special cases (preliminary report). In *Working Notes of IJCAI-05 Workshop on Nonmonotonic Reasoning, Action, and Change*, pages 34–38, 2005.

[Erdoğan and Lifschitz, 2006] Selim T. Erdoğan and Vladimir Lifschitz. Actions as special cases. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 377–387, 2006.

[Erdoğan, 2008] Selim T. Erdoğan. *A Library of General-Purpose Action Descriptions*. PhD thesis, University of Texas at Austin, 2008.

[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 Inclezan, 2009] Michael Gelfond and Daniela Inclezan. Yet another modular action language.[5] Unpublished draft, 2009.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen,

[5] `http://www.krlab.cs.ttu.edu/papers/download/gi09.pdf` .

editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[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.[6] *Electronic Transactions on Artificial Intelligence*, 3:195–210, 1998.

[Gelfond, 2006] Michael Gelfond. Going places — notes on a modular development of knowledge about travel.[7] In *Working Notes of the AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, 2006.

[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.

[6] `http://www.ep.liu.se/ea/cis/1998/016/` .
[7] `http://www.krlab.cs.ttu.edu/papers/ download/gel06.pdf` .

[Gustafsson and Kvarnström, 2004] Joakim Gustafsson and Jonas Kvarnström. Elaboration tolerance through object-orientation. *Artificial Intelligence*, 153(1–2):239–285, 2004.

[Ianni *et al.*, 2004] Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, Maria Carmela Santoro, and Francesco Calimeri. Enhancing answer set programming with templates. In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)*, pages 233–239, 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 *et al.*, 2009] Joohyung Lee, Yuliya Lierler, Vladimir Lifschitz, and Fangkai Yang. Expressing synonymity in causal logic and in logic programming.[8] Unpublished draft, 2009.

[Lee, 2005] Joohyung Lee. *Automated Reasoning about Actions.*[9] PhD thesis, University of Texas at Austin, 2005.

[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

[8] `http://www.cs.utexas.edu/users/vl/tmp/lf24.pdf` .
[9] `http://www.cs.utexas.edu/users/appsmurf/papers/dissertation.ps` .

description language. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 853–859, 2006.

[Lifschitz and Ren, 2007] Vladimir Lifschitz and Wanwan Ren. The semantics of variables in action descriptions. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 1025–1030, 2007.

[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, 1994] Vladimir Lifschitz. Circumscription. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1994.

[Lifschitz, 1997] Vladimir Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.

[Lifschitz, 1999] Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[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, 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.

[Marek and Truszczyński, 1999] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[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, 1997] Norman McCain. *Causality in Commonsense Reasoning about Actions.*[10] PhD thesis, University of Texas at Austin, 1997.

[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, 1980] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39,171–172, 1980.

[McCarthy, 1986] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.

[McCarthy, 1987] John McCarthy. Generality in Artificial Intelligence. *Communications of ACM*, 30(12):1030–1035, 1987. Reproduced in [McCarthy, 1990].

[10] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.gz` .

[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.[11] In progress, 2007.

[Niemelä and Simons, 2000] Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

[Niemelä, 1999] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[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.

[Pednault, 1994] Edwin Pednault. ADL and the state-transition model of action. *Journal of Logic and Computation*, 4:467–512, 1994.

[Shanahan, 1997] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[Turner, 1997] Hudson Turner. Representing actions in logic programs and default

[11] `http://www-formal.stanford.edu/jmc/elaboration.html` .

theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

[Van Gelder *et al.*, 1991] Allen Van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[Veloso, 1992] Manuela Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Carnegie Mellon University, 1992. PhD thesis.

Vita

Wanwan Ren was born in Shangqiu, China, in 1975. He received the B.S. degree from Tsinghua University in 1995 and the M.S. degree from Peking University in 1998, both in Physics. Since August 1998 he has been studying at the graduate school at The University of Texas at Austin, first in the Department of Physics, then in the Department of Computer Sciences from August 2000. He received the M.S. degree in Computer Sciences from The University of Texas at Austin in August 2002. From January 2003 onwards, he has been enrolled in the doctoral program in Computer Sciences at The University of Texas at Austin.

Permanent Address: College of Continuing Education
Shangqiu Normal College
Shangqiu, Henan, 476000
P. R. China

This dissertation was typeset with LaTeX 2_ε by the author.