The Dissertation Committee for Joohyung Lee

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

Automated Reasoning about Actions

Committee:

Vladimir Lifschitz, Supervisor

Robert S. Boyer

Bruce W. Porter

Peter Stone

Hudson Turner

Automated Reasoning about Actions

by

Joohyung Lee, B.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

May 2005

To my family

Acknowledgments

Deepest thanks from the bottom of my heart to my advisor, Vladimir Lifschitz for his continuous support and encouragement throughout this work. His way of conducting research and caring others is exemplary that I will continue to strive to emulate. Thanks also to Elena Lifschitz for valuable advice and care for my wife and me.

I am thankful to the other members of the committee: Robert S. Boyer, Bruce W. Porter, Peter Stone, and Hudson Turner for careful reading of the dissertation and useful comments on it.

I have benefited from many useful discussions with teachers and friends. Thanks to all of them including Chitta Baral, Jonathan Campbell, Esra Erdem, Selim Erdoğan, Paolo Ferraris, Michael Gelfond, Yuliya Lierler, Fangzhen Lin, Marco Maratea, Wanwan Ren and Hudson.

I have relied on Yoonsuck Choe, Wongeun Chung, Yang-Suk Kee, Seungchan Kim, Roberto E. Lopez-Herrejon and Jungkun Seo when I needed to make difficult decisions, and I am grateful to them for their advice. I am also thankful to Guru Huchachar, Eunjin Jung, Madhusudan Kayastha, Hyunok Oh, Chun-Yen Wang, the members of the Texas Action Group at Austin, friends from the department of

computer sciences, friends from the Korean Baptist Church of Austin, and friends from Korea for their support and friendship.

I was partially supported by a fellowship from the Korea Foundation for Advanced Studies, and I am thankful for it. My research was also partially supported by NSF under Grant IIS-9732744 and Grant IIS-0412907, and the Texas Higher Education Coordinating Board under Grant 003658-0322-2001.

I thank my God who loves me without ever stopping. I thank my parents Soo-Ung Lee and Chun-Hee Kim, my parents-in-law Jin Gyu Park and Sun Hee Suh, my brother Changhyung Lee and my sister-in-law Chae Yon Park for their love and support. My deepest love goes to my wife with whom I share everything and our first baby who is yet in his mom's womb.

JOOHYUNG LEE

The University of Texas at Austin

May 2005

Automated Reasoning about Actions

Publication No. _____

Joohyung Lee, Ph.D.

The University of Texas at Austin, 2005

Supervisor: Vladimir Lifschitz

The study of reasoning about actions is an important subarea of the theory of commonsense reasoning. It is concerned with developing appropriate systems of logic for describing actions and their effects on the world. In spite of the fact that this reasoning is based on common sense and does not involve any specialized knowledge, attempts to formalize it using classical logic encountered serious difficulties, which have led to the emergence of a new field, *nonmonotonic logics*.

In particular, McCain and Turner introduced the causal logic in which the notions of "being caused" and "being true" are distinguished. Based on their logic, Giunchiglia and Lifschitz proposed a high level action language $\mathcal{C}$, which is a formal model of parts of natural language that are used for describing properties of actions. The causal logic and $\mathcal{C}$, along with the concept of satisfiability planning, provided us with a widely applicable and efficient method of automated reasoning about actions, which led to the creation of the Causal Calculator (CCalc).

In this dissertation, we have identified several essential limitations of the McCain–Turner causal logic and action language $\mathcal{C}$. To overcome these limitations, we defined an extension of the causal logic to multi-valued formulas and a new action language $\mathcal{C}+$. Language $\mathcal{C}+$ can represent non-propositional fluents, defined fluents, additive fluents, rigid constants, and defeasible causal laws. Second, we have redesigned and reimplemented CCALC to account for these extensions, and tested the new CCALC and the underlying theory by applying them to several new, more difficult examples of commonsense reasoning. The input language of the new CCALC is more elaboration tolerant than the old version. Last, we have shown how to turn causal logic into propositional logic based on the idea of "loop formulas" that originated from logic programming under the answer set semantics.

Table of Contents

List of Figures

Chapter 1

Introduction

For a long time humans have been extending their abilities via their own inventions. Mechanical devices have been developed to fulfill part of the dream. Ever since computers were first built, the dream has geared its way to more intelligent tasks. Once a task was well studied to automate, the use of computers became essential.

As we learn how to build systems for doing such tasks, computers seem to become more "intelligent." However, there are many human abilities that still cannot be automated using the knowledge that we have: how can we build a system that can understand and speak a natural language as well as a human (*The Natural Language Problem*), how can we build a system that can see as well as a human (*The Vision Problem*), to list a few.

The ability to reason is also one of them. The intellectual mechanisms involved in reasoning are not well understood, even[1] in the cases when reasoning is based on common sense and does not involve any specialized knowledge. Indeed, everyday life is full of commonsense problems, but a human has no diffi-

[1] Or one might say, especially.

culty solving them. However, we have little idea how a human's reasoning mechanism works, let alone how to automate it. Even a simple-minded person can easily devise a commonsense problem that would be a considerable challenge to researchers in this area. For instances of commonsense problems that have particularly interested researchers, one may consult the Common Sense Problem Page (`http://www-formal.stanford.edu/leora/cs`). A monograph by Davis [1990] contains a survey of various topics in this area.

For instance, the following are a few instances of problems we want to solve automatically:

- **Monkey and Bananas** There is a monkey in a room that contains a box and a bunch of bananas hanging from the ceiling. The bananas are beyond his reach, but if he climbs onto the box, he would be able to grasp it. How can a monkey grasp the bananas?

- **Missionaries and Cannibals** Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross?

- **Getting to the Airport** I am seated at my desk at home and my car is at home also. How can I get to the airport [McCarthy, 1959]? [2]

Many AI researchers have been trying to endow computers with intelligence through formal logic. However, their first attempts were not successful because theories based on classical logic were not adequate for solving commonsense prob-

[2] This is the oldest planning problem in the AI literature.

lems. It was a new challenge that logicians had been ignorant of, but one that AI researchers had to confront to fulfill their dream.

One of the most fundamental difficulties was that all systems of logics known at the time were *monotonic*: if a conclusion is derivable from a set of axioms, then it is still derivable even after adding more axioms. We may use the same old derivation which does not include additional axioms. Monotonicity is natural in usual mathematics. However, it is not desirable in formalizing commonsense reasoning, where a conclusion may no longer be derivable when we add new assumptions. For instance, a conclusion that is based on assumptions such as "*normally*, the car is drivable" may be retracted later under certain exceptional circumstances such as "there is no gas in the car," and then we may get a totally different conclusion. This may once again be retracted if we are told that "the car is run by electricity, and it has enough of it." Still the new conclusion can be retracted once again if we are told that "it is a toy car." It appears that one can continue to build an arbitrarily long sequence of exceptions to any commonsense conclusion.

Despite this fact, humans have no difficulty drawing a conclusion. In a sense, humans' reasoning may involve jumping to a conclusion. For instance, when we hear that there is a car in the garage, we jump to a conclusion that the car can be used to drive. Such a conclusion can be retracted in the presence of additional information that defeats the assumptions on which the conclusion was based. Logics that have this property are called *nonmonotonic logics* and they were proposed by AI researchers in the early 1980's. The formalism we propose in this dissertation is also nonmonotonic.

Although significant progress has been made in the last decade, the theory

of commonsense reasoning is still far from being complete. In this dissertation, we focus on the subarea called *reasoning about actions*, in which we are concerned with the formalization and automation of reasoning about the effects of actions. By an action we mean anything that can be executed, and then may affect the state of the world. In fact, one can see that all three examples above involve actions. These actions are

- walking, pushing the box, climbing onto the box, and grasping the bananas
- crossing the river
- walking and driving

respectively. Walking changes the location of the monkey; climbing onto the box changes the status of being on the box; crossing the river affects the number of people on each bank, etc.

The automation of commonsense reasoning about actions is the subject of this dissertation. Our work is based on a few successes in the last decade. In particular, McCain and Turner introduced a nonmonotonic causal logic [McCain and Turner, 1997], in which the notions of "being caused" and "being true" are distinguished. Based on it, Giunchiglia and Lifschitz proposed a high level action language $\mathcal{C}$, which is a formal model of parts of natural language that are used for describing properties of actions. The causal logic and $\mathcal{C}$, along with the concept of satisfiability planning, provided a widely applicable and efficient method of automated reasoning about actions, which led to the creation of the Causal Calculator (CCalc).

In this dissertation, we have identified several essential limitations of the McCain–Turner causal logic and action language $\mathcal{C}$. To overcome these limitations,

we defined an extension of the causal logic to multi-valued formulas and a new action language $\mathcal{C}+$. Language $\mathcal{C}+$ can represent non-propositional fluents, defined fluents, additive fluents, rigid constants, and defeasible causal laws. Second, we have redesigned and reimplemented CCALC to account for these extensions, and tested the new CCALC and the underlying theory by applying them to several new, more difficult examples of commonsense reasoning. Last, we have shown how to turn causal logic into propositional logic based on the idea of "loop formulas" that originated from logic programming under the answer set semantics.

After reviewing earlier work on the formalization and automation of reasoning about actions in Chapters 2–4, we discuss the need to extend the McCain–Turner causal logic, language $\mathcal{C}$ and an early version of CCALC in Chapter 5. In Chapter 6 we present an extension of the McCain–Turner causal logic called multi-valued causal logic, a new action language $\mathcal{C}+$, and the new version of CCALC that overcome the limitations, and relate $\mathcal{C}+$ to the language ADL from [Pednault, 1994]. In Chapter 7 we test expressive possibilities of $\mathcal{C}+$ and CCALC by formalizing an action domain of nontrivial size. We identify a class of fluents that we call *additive* and show how $\mathcal{C}+$ can be used to talk about the effects of actions on such fluents in Chapter 8. In Chapter 9 we formalize McCarthy's elaborations of the Missionaries and Cannibals Puzzle in the language of the new CCALC. We show how to turn causal logic into propositional logic using the idea of loop formulas in Chapter 10, and apply loop formulas to the problem of splitting a causal theory in Chapter 11.

Chapter 2

Background

In his classic paper [McCarthy, 1959], McCarthy proposed to create a software system that he called the *advice taker*. The system is supposed to draw relevant conclusions from the set of premises, mainly in the form of declarative sentences, describing a domain of consideration. If the information stored in the system needs to be changed, extended or deleted, that should be done by just updating the premises, rather than by rewriting the system's internal code. Moreover, heuristics should also be introduced by declarative sentences. The airport problem mentioned in Chapter 1 was the example used in the paper to explain this idea. The system is expected to generate the plan of getting to the airport given a declarative description of the problem.

The idea of the advice taker was new, and there were many details to be clarified; many serious difficulties were identified later. In the course of discussion, Bar-Hillel commented, "Dr. McCarthy's paper belongs in the Journal of Half-Baked Ideas." Even now, more than 40 years later, the idea is still being baked. However,

we have seen much progress. Recently, CCALC was applied to solving the airport problem [Lifschitz *et al.*, 2000]. In this chapter, we present how the research in this area has evolved.

2.1 Problems in Formal Reasoning about Actions

It seems natural to choose formal logic as a vehicle for representing commonsense knowledge due to its precise and declarative semantics. Hayes [1977] pointed out that a logical model theory provides accounts for the meaning of a representation or representational language and helps us compare different representations or languages. Researchers hoped that a computer would be able to derive relevant conclusion from properly axiomatized knowledge.

But soon serious difficulties with formal logic were recognized. Some of the problems were due to the implausible number of axioms that were required. The most important one is the *frame problem*, which was first identified in [McCarthy and Hayes, 1969]. The problem is how to represent what remains *unchanged* after executing an action. Axiomatizers have to describe not only the things that change, but also the things that do not change; without that, one would not be able to draw many useful conclusions. The difficulty is that, in commonsense domains, there are too many things that do not change, and enumerating all of them would not be feasible (it looks also non-commonsensical to have to enumerate them all). For instance, when we describe an action of walking to the car, we also need to list all things that do not move: the desk, the car, the airport, the house and so on.

The frame problem becomes more difficult in the presence of indirect effects of an action. The problem of describing indirect effects of an action is called the

ramification problem [Finger, 1986]. For instance, if I drive to the airport, not only my location and the location of the car change, but also the locations of things in my pocket and the trunk change. Enumerating all indirect effects is also tedious.

2.2 Nonmonotonic Reasoning

It was observed that the difficulties with formal logic described above are related to the fact that classical logic is *monotonic*: for any sets of premises A and B such that $A \subseteq B$, if a sentence F follows from A, then F follows from B also. In other words, every conclusion that can be derived from A is also derivable from B. This is not desirable in commonsense reasoning: as discussed in the introduction, when additional assumptions are made, some of the conclusions may need to be retracted. This was a challenge to AI researchers, and several systems of nonmonotonic reasoning were invented in response.

A 1980 issue of the journal of *Artificial Intelligence* presented three forms of nonmonotonic reasoning: circumscription by McCarthy [1980], default logic by Reiter [1980], and a nonmonotonic logic by McDermott and Doyle [1980]. The concept of circumscription was extended in [McCarthy, 1986], and an influential modal nonmonotonic logic called autoepistemic logic was introduced by Moore [1985].

Every system of nonmonotonic reasoning provides a method for representing "defaults." One particularly important default is *the commonsense law of inertia*, which says that everything tends to remain as it was. Formalizing this idea was recognized as a key to solving the frame problem.

While the earlier forms of nonmonotonic reasoning were going through refinements and improvements, in 1987, Hanks and McDermott challenged the re-

search community by arguing that formal logic is no good for representing commonsense knowledge. As an example, they presented the so-called "Yale shooting problem" [Hanks and McDermott, 1987], where McCarthy's revised form of circumscription [McCarthy, 1986] could not account for a simple fact.

> There is a gun and a person (in some versions, a turkey) whose name is Fred. If the gun is loaded, shooting it kills Fred. Now consider the following scenario. Initially Fred was alive, and the gun was not loaded. Next the gun is loaded, and after waiting, the gun is shot. Is Fred dead?

Intuitively, the answer should be yes. However, McCarthy's 1986 proposal could not justify this. It left open the possibility that the gun gets unloaded by itself during the execution of the wait action.

The failure discouraged some AI researchers and made them abandon the logicist approach to commonsense reasoning. But others continued to extend the systems of logic and came up with various solutions in response to the challenge. Some of them are [Lifschitz, 1987], [Morris, 1988], [Gelfond, 1989], [Baker, 1991] and [Lifschitz, 1991].

Logic programming became a member of the family of nonmonotonic reasoning systems once the semantics of "negation as failure" was clarified. Among the semantics, influential are the completion semantics [Clark, 1978], the well-founded semantics [Van Gelder *et al.*, 1991], and the stable model or the answer set semantics [Gelfond and Lifschitz, 1988]. Gelfond [1987] showed how to translate logic programs into autoepistemic logic. Solutions to the Yale Shooting problem using logic programs are described in [Eshghi and Kowalski, 1989], [Evans, 1989], [Apt and Bezem, 1990].

2.3 Nonmonotonic Theories of Causality

Causality has been a major subject of study by philosophers from the ancient times [1], and now it is studied in AI as well.

In the natural sciences, the distinction between a material implication ("If A holds, then B holds") and a causal relation ("A causes B") is commonly disregarded. Such distinction, however, turned out to be quite useful in commonsense reasoning. As a result, nonmonotonic theories based on causality received considerable attention.

Pearl [1988] investigated the distinction between causal and non-causal grounds in general default reasoning. Geffner [1990] introduced a modal operator for representing causality. Lin [1995] introduced the predicate *Caused*; his proposal made it possible to conveniently express the indirect effects of an action, as well as the direct effects, using circumscription.

Later, McCain and Turner [1997] introduced a causal logic in which the notions of "being caused" and "being true" are distinguished using expressions of the form

$$F \Leftarrow G \tag{2.1}$$

where F and G are propositional formulas. Intuitively (2.1) is understood as the assertion that F is caused if G holds. The semantics of the causal logic is based on "the principle of universal causation," which says that every fact that obtains is caused. This strong philosophical commitment is rewarded by the mathematical simplicity in the semantics. Universal Causal Logic (UCL) [Turner, 1999] extends

[1] Aristotle enumerated four kinds of causes: the material, the formal, the efficient, and the final. Rene Descartes, David Hume, Immanuel Kant, and John Stuart Mill were also among the philosophers who studied causality.

the language of causal theories to a modal framework. Although the syntax of Geffner's theory and UCL are similar, their semantics are not, and there seems to be no precise relationship between them. The semantics of McCain and Turner's causal logic is closely related to that of logic programming under the answer set semantics.

The systems proposed by Geffner, Lin, McCain and Turner allow us to express "static causal laws"—causal dependencies between fluents. This is essential for solving the ramification problem.

2.4 Action Languages

Action languages [Gelfond and Lifschitz, 1998] are formal models of parts of natural language that are used for describing the effects of actions. They define "transition systems"—directed graphs whose vertices correspond to states and whose edges are labeled by actions. Originally, action languages were developed to represent the properties of actions in a high level notation. Their simple but concise syntax helps us compare them and improve our understanding of reasoning about actions.

The STRIPS language [Fikes and Nilsson, 1971] is not an action language in the sense of [Gelfond and Lifschitz, 1998], but is closely related. Despite its limited expressivity and semantic pitfalls [Lifschitz, 1987], STRIPS's influence has been significant for two reasons: the language provides a built-in solution to the frame problem; efficient computation can be carried out by employing a resolution theorem prover in finding a sequence of STRIPS operators that leads to a world model in which a given goal formula is true.

Many extensions that improve the expressive power of STRIPS were pro-

posed. Pednault's ADL [Pednault, 1994] extended STRIPS by allowing symbols for non-propositional fluents and conditional effects of actions. Gelfond and Lifschitz [1993] introduced language $\mathcal{A}$ (which is essentially the propositional fragment of ADL) and related it to logic programming. Similar results for a language that permits the concurrent execution of actions were proved in [Baral and Gelfond, 1997], and for a language with static causal laws in [Turner, 1997]. That work, along with the theory of nonmonotonic causal reasoning presented in [McCain and Turner, 1997], has led to the design of language $\mathcal{C}$ [Giunchiglia and Lifschitz, 1998], which is a basis of the action language $\mathcal{C}+$ that we present in this dissertation.

2.5 Elaboration Tolerance

McCarthy [1998] expressed the view that human-level AI would require what he called elaboration tolerance:

> A formalism is elaboration tolerant to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances. Representations of information in natural language have good elaboration tolerance when used with human background knowledge. Human-level AI will require representations with much more elaboration tolerance than those used by present AI programs, because human-level AI needs to be able to take new phenomena into account.
>
> The simplest kind of elaboration is the addition of new formulas. Next comes changing the values of parameters. Adding new arguments to

functions and predicates represents more of a change.

In the paper McCarthy illustrated the idea by defining 19 variants of the Missionaries and Cannibals Puzzle (MCP). Here are some of his elaborations:

- The boat can carry three.
- There is an oar on each bank.
- Only one missionary and one cannibal can row.
- The biggest cannibal cannot fit in the boat with another person.
- If the biggest cannibal is isolated with the smallest missionary, the latter will be eaten.
- Three missionaries along with a cannibal can convert him into a missionary.
- There is a bridge.
- The boat leaks and must be bailed concurrently with rowing.
- There is an island.
- There are four cannibals and four missionaries, but if the strongest of the missionaries rows fast enough, the cannibals won't have gotten so hungry that they will eat the missionaries.

When humans are told about the elaborations above, they understand the changes using their background knowledge expressed in natural language without having to start from scratch.

Lifschitz [2000] showed how to formalize the ten elaborations of MCP above in the language of CCALC. Instead of formalizing each elaboration from scratch, he "factored out" their common part; each formalization of an elaboration does not modify the common part, but just adds to it a few propositions that express the change. This is the simplest kind of elaboration that McCarthy discussed. CCALC has determined the shortest number of steps to solve each elaboration and showed the solution.

2.6 SAT solvers

SATISFIABILITY (or SAT for short) is the problem of determining whether a given Boolean expression in conjunctive normal form is satisfiable. This is the first problem proven to be NP-complete [Cook, 1971].

Systems that solve instances of this problem are called SAT solvers. Many of them are based on an algorithm due to Davis, Logemann and Loveland [1962]. Various techniques such as intelligent backtracking, learning, backjumping and a rapid restart strategy have been used to improve the efficiency of SAT solvers. At the time of this writing, hundreds of thousands of atoms, and millions of clauses can be handled reasonably well in many cases.

Since various problems can be cast as propositional theories, SAT solvers are widely applied. In "satisfiability planning" [Kautz and Selman, 1992] a planning problem is encoded as a propositional theory so that a model of the theory corresponds to a plan—a sequence of actions—that leads to a goal state from an initial state. The plan can be found by running a SAT solver. Blackbox [2] is a

[2] `http://www.cs.washington.edu/homes/kautz/blackbox` .

planning system that converts a STRIPS formalization of a planning problem into a propositional theory, and then finds its models using SAT solvers.

SAT solvers have many applications to areas other than reasoning about action. For instance, they have been applied to the formal verification of hardware systems with emphasis in Bounded Model Checking: NuSMV2 [3] is a SAT-based symbolic model checker that turned out to be more efficient than BDD-based NuSMV; GrAnDe [4] is a theorem prover based on SAT solvers; SAT solvers have been also used for finding attacks to a set of well-known authentication protocols [Armando and Compagna, 2002].

Some SAT solvers are complete, that is, they find a model if there exists one, and answer "no" if there is none. Others sacrifice completeness in return for efficiency. Most SAT solvers today accept the DIMACS input format. This simplifies the efforts required to test and compare the solvers. Also systems employing SAT solvers as their search engines have the flexibility of choosing different solvers. For instance, one can run an incomplete solver first, and if it does not terminate after certain time, run a complete solver. Competitions for SAT solvers are held frequently to encourage the creation of more efficient systems.[5]

Carefully engineered solvers have shown significant speed-up. Chaff [Moskewicz *et al.*, 2001; Zhang *et al.*, 2001] was designed from the beginning to handle large formulas from a very specific area (mostly Bounded Model Checking) using "lazy" data structures, and also integrated a new form of learning, taking advantage of the overall lazy data structures used. Chaff outperforms existing SAT solvers on a

[3] `http://nusmv.irst.itc.it/` .

[4] `http://www.cs.miami.edu/~tptp/ATPSystems/GrAnDe/` .

[5] For a recent report, consult `http://www.satisfiability.org/SAT04/` .

large set of "structured" (as opposed to random) instances. This efficiency boost is expected to make SAT solvers more widely applicable.

Chapter 3

Logic Programs and the McCain–Turner Causal Logic

The underlying nonmonotonic formalism we choose for formalizing the properties of actions is causal logic. It is closely related to the answer set semantics (also known as the stable model semantics) of logic programs by Gelfond and Lifschitz [1988], which has led to a new declarative programming paradigm called answer set programming [Lifschitz, 1999; Marek and Truszczyński, 1999; Niemelä, 1999].

A special case of the answer set semantics is closely related to a simple nonmonotonic formalism called Clark's completion [Clark, 1978]. Completion is attractive because it is defined as a transformation of logic programs to classical logic, but it sometimes gives unintuitive results [Przymusinski, 1989, Section 4.1]. The concept of completion was extended to causal logic by McCain and Turner [1997], and the relationship between the semantics of causal logic and completion turned out to be more immediate than the relationship between the answer set semantics

and completion. This idea has led to an efficient implementation of automated reasoning about actions.

In this chapter we review the answer set semantics and the semantics of causal logic, and their relationships with completion.

3.1 Answer Set Semantics for Normal Programs

We review the answer set semantics for normal programs [Gelfond and Lifschitz, 1988]. The word *atom* is understood here as in propositional logic.

A *(normal) rule* is an expression of the form

$$p_1 \leftarrow p_2, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n \tag{3.1}$$

$(1 \leq m \leq n)$ where all p_i are atoms. Atom p_1 is called the *head*, and the part

$$p_2, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$

is called the *body* of the rule. We will often write (3.1) in the form

$$p_1 \leftarrow B, F \tag{3.2}$$

where B is $p_2, \ldots, p_m$, and F is $not\ p_{m+1}, \ldots, not\ p_n$, and we will sometimes identify B with the set $\{p_2, \ldots, p_m\}$. If the body is empty, then $\leftarrow$ can be dropped.

A *(normal logic) program* is a finite set of rules of form (3.1).[1]

We say that a set X of atoms *satisfies* the body B, F of rule (3.2) (symbolically, $X \models B, F$) if $p_2, \ldots, p_m \in X$ and $p_{m+1}, \ldots, p_n \notin X$. We say that X *satisfies* a normal program Π (symbolically, $X \models \Pi$) if, for every rule (3.2) of that program, $p_1 \in X$ whenever X satisfies B, F.

[1] In the literature, programs are allowed to contain infinitely many rules, but in this dissertation, for simplicity, we restrict attention to finite programs only.

Answer sets are defined by a fixpoint definition. The *reduct* Π^X of a normal program Π with respect to a set X of atoms is obtained from Π by

- deleting each rule (3.2) such that $X \not\models F$, and
- replacing each remaining rule (3.2) by $p_1 \leftarrow B$.

A set X of atoms is an *answer set* of Π if X is minimal among the sets of atoms that satisfy Π^X.

For example, consider the following program Π_1:

$$p \leftarrow not\ q$$

$$q \leftarrow not\ p$$

Consider, one by one, all sets formed from the atoms p and q:

- $X_1 = \emptyset$. The reduct $\Pi_1^{X_1}$ is $\{p, q\}$, which X_1 does not satisfy. Consequently, X_1 is not an answer set of Π_1.
- $X_2 = \{p\}$. The reduct $\Pi_1^{X_2}$ is $\{p\}$. Since X_2 is minimal among the sets of atoms that satisfy the reduct, X_2 is an answer set of Π_1.
- $X_3 = \{q\}$. Similarly to the above, the reduct $\Pi_1^{X_3}$ is $\{q\}$. Since X_3 is minimal among the sets of atoms that satisfy $\Pi_1^{X_3}$, X_3 is an answer set of Π_1.
- $X_4 = \{p, q\}$. The reduct $\Pi_1^{X_4}$ is $\emptyset$, which X_4 satisfies, but it is not minimal among the sets of atoms that satisfy the reduct. Consequently, X_4 is not an answer set of Π_1.

Thus we see that X_2 and X_3 are the only answer sets of Π_1.

3.2 Completion

Let Π be a program whose rules have the form (3.2). The completion of Π, $Comp(\Pi)$, consists of the equivalences

$$p_1 \equiv \bigvee_{p_1 \leftarrow B, F \in \Pi} B \wedge F \tag{3.3}$$

for all atoms p_1 that occur in Π.[2]

For example, $Comp(\Pi_1)$ is

$$p \equiv \neg q$$

$$q \equiv \neg p,$$

whose models are $\{p\}$, $\{q\}$, which are the same as the answer sets of Π_1.

Proposition [Erdem and Lifschitz, 2003, Proposition 1] *For any program* Π *and any set* X *of atoms, if* X *is an answer set of* Π *then* X *is a model of* $Comp(\Pi)$.

It is well known that the converse of this proposition does not hold. The one-rule program $p \leftarrow p$ is a standard counterexample; both $\emptyset$ and $\{p\}$ are the models of $Comp(\Pi)$, but only $\emptyset$ is the answer set of Π.

Fages [1994] showed that if a program is "tight," then the converse of the proposition holds as well. Erdem and Lifschitz [2003] generalized Fages' theorem and extended it to a more general class of programs.

3.3 The McCain–Turner Causal Logic

Like logic programs, causal theories consist of rules, but they are different in that heads and bodies are arbitrary formulas in propositional logic. In this sense they are

[2] Completion defined here can easily be extended to the case where rules are allowed to have empty heads, which is omitted here for simplicity.

more "propositional logic friendly" than logic programs. In this section we review the semantics of causal logic.

A *propositional signature* is a set of symbols of propositional atoms. A *formula* is a propositional combination of atoms as in propositional logic. An *interpretation* of σ is a function that maps each element of σ to the truth values.

By a *(causal) rule* we mean an expression of the form

$$F \Leftarrow G$$

("F is caused if G holds"), where F, G are formulas in propositional logic of the signature σ. Formula F is called the head and G is called the body of the rule. Rules with the head $\bot$ are called *constraints*.

A *causal theory* is a finite set of causal rules.

Like the semantics of a logic program, the semantics of a causal theory is given by a fixpoint definition. Let T be a causal theory, and I 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.

Intuitively, T^I is the set of formulas that are caused, according to the rules of T, under interpretation I. If this set has no models or more than one model, then, according to the definition above, I is not considered a model of T. If T^I has exactly one model, but that model is different from I, then I is not a model of T either. The only case when I is a model of T is when I satisfies every formula in the reduct, and no other interpretation does.

If a causal theory T has a model, we say that it is *consistent*, or *satisfiable*. If every model of T satisfies a formula F then we say that T *entails* F and write

$T \models F$.

As an example, take the following causal theory T_1 whose signature is $\{p, q\}$:

$$\begin{aligned} p &\Leftarrow q \\ q &\Leftarrow q \\ \neg q &\Leftarrow \neg q. \end{aligned} \tag{3.4}$$

Consider, one by one, all interpretations of that signature (we identify an interpretation with the set of literals that are true in it):

- $I_1 = \{p, q\}$. The reduct consists of the heads of the first two rules of T_1: $T_1^{I_1} = \{p, q\}$. Since I_1 is the unique model of $T_1^{I_1}$, it is a model of T_1.
- $I_2 = \{\neg p, q\}$. The reduct is the same as above, and I_2 is not a model of the reduct. Consequently, I_2 is not a model of T_1.
- $I_3 = \{p, \neg q\}$. The only element of the reduct is the head of the third rule of T_1: $T_1^{I_3} = \{\neg q\}$. It has two models. Consequently, I_3 is not a model of T_1.
- $I_4 = \{\neg p, \neg q\}$. The reduct is the same as above, so that I_4 is not a model of T_1 either.

Thus we see that I_1 is the only model of T_1.

Consider another example T_2 whose signature is again $\{p, q\}$:

$$\begin{aligned} p \vee \neg q &\Leftarrow \top \\ \neg p \vee q &\Leftarrow \top. \end{aligned}$$

The reduct T_2^I is equal to the set of the heads of the rules in T_2 regardless of the interpretation I, so that it has two models, $\{p, q\}$ and $\{\neg p, \neg q\}$. Therefore, T_2 has no models.

T_3 is the following theory of the same signature that adds one rule to T_2:

$$p \vee \neg q \Leftarrow \top$$
$$\neg p \vee q \Leftarrow \top$$
$$p \vee q \Leftarrow \top.$$

Similarly to the previous example, T_3^I is equal to the set of the heads of the rules in T_3 regardless of the interpretation I. Now $\{\neg p, \neg q\}$ is not a model of T_3^I, so that T_3 has one model: $\{p, q\}$.

Theories T_2 and T_3 illustrate the nonmonotonicity of causal logic: we may get a new model by adding more rules.

3.4 Literal Completion

A causal theory is called *definite* if the head of every rule in it is either a literal or $\bot$. For a definite theory, we can describe its models in terms of "literal completion" [McCain and Turner, 1997], which is similar to Clark's completion for normal logic programs.

Consider a definite causal theory T of a signature σ. For each literal l, the *literal completion formula* for l is the formula

$$l \equiv G_1 \vee \cdots \vee G_n$$

where $G_1, \ldots, G_n$ $(n \geq 0)$ are the bodies of the rules of T with head l. The *(literal) completion* of T is obtained by taking the completion formulas for every literal of σ, along with the formula $\neg F$ for each constraint $\bot \Leftarrow F$ in T.

For example, the completion of T_1 is

$$\begin{aligned} p &\equiv q \\ \neg p &\equiv \bot \\ q &\equiv q \\ \neg q &\equiv \neg q, \end{aligned} \tag{3.5}$$

and its only model is $\{p, q\}$, which is exactly the model found above using the definition of causal logic.

The relationship between causal logic and completion is more immediate than the relationship between logic programs and completion described in Proposition 1 from [Erdem and Lifschitz, 2003] (Section 3.2):

Proposition *[McCain and Turner, 1997] The models of a definite causal theory are precisely the models of its completion.*

However, the method of completion is not applicable to nondefinite theories, such as T_2 and T_3.

Here are two more examples of the use of completion. First, we will show how to turn any set Γ of formulas into a causal theory that has the same models as Γ. The rules of this theory are

- $l \Leftarrow l$ for every literal l of σ, and
- the constraints $\bot \Leftarrow \neg F$ for every $F \in \Gamma$.

The completion of this theory consists of the formulas $l \equiv l$ for all literals l and the formulas $\neg\neg F$ for all $F \in \Gamma$. Clearly, the completion is equivalent to Γ.

Second, definite theories can be used to express the "closed-world assumption," [Reiter, 1978] as follows. Take a signature σ. The assumption that the elements of σ are false by default can be expressed by the rules

$$\neg a \Leftarrow \neg a \qquad (a \in \sigma) \tag{3.6}$$

(if a is false then there is a cause for this). If, for some subset S of σ, we combine (3.6) with the rules

$$a \Leftarrow \top \qquad (a \in S),$$

we will get a causal theory whose only model is the interpretation I that assigns **t** to the atoms in S and **f** to all other atoms. Indeed, the completion of this theory consists of the formulas

$$\begin{aligned} a &\equiv \top \qquad (a \in S), \\ a &\equiv \bot \qquad (a \in \sigma \setminus S), \\ \neg a &\equiv \neg a \qquad (a \in \sigma), \end{aligned}$$

and I is the only model of these formulas.

The proposition above shows that the satisfiability problem for definite causal theories belongs to class NP. It is clearly NP-complete.

3.5 The Causal Calculator (CCalc)

The proposition from Section 3.4 tells us that the models of definite theories can be computed by SAT solvers. This idea led McCain to design the Causal Calculator (CCalc) [3]—an implementation of definite causal theories. Computationally, CCalc turns a definite theory into a propositional theory by literal completion,

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

and then calls SAT solvers to find the models of the propositional theory, which, in turn, correspond to the models of the causal theory.

The original version of CCALC was implemented in Prolog as part of McCain's dissertation [McCain, 1997]. The idea is similar to satisfiability planning [Kautz and Selman, 1992] but the formalism of CCALC is much more expressive than the STRIPS based formalisms [McCain and Turner, 1998]. An early version of CCALC was applied to formalizing several challenge problems in the theory of commonsense knowledge, including McCarthy's airport example [Lifschitz *et al.*, 2000] and elaborations of the Missionaries and Cannibals Puzzle [Lifschitz, 2000].

We will talk about CCALC in more detail in the following chapter.

Chapter 4

Action Language $\mathcal{C}$ and the Causal Calculator

4.1 Language $\mathcal{C}$

The review of $\mathcal{C}$ in this section follows [Giunchiglia and Lifschitz, 1998].

4.1.1 Syntax

In $\mathcal{C}$, a signature σ is partitioned into two groups of symbols: *fluent symbols* σ^{fl} and *action symbols* σ^{act}. A *fluent formula* is a formula that does not contain action symbols.

Consider the monkey and bananas problem described in Chapter 1. To formalize the problem in a declarative language, one needs to be able to describe

- the locations of the monkey, the bananas, and the box,
- whether the monkey is on the box, and

- whether the monkey has the bananas.

Assuming that the possible locations of the monkey, the bananas, and the box are L_1, L_2, L_3, a signature that would allow us to talk about the states consists of symbols:

$$\begin{array}{l} At(x,l) \qquad (x \in \{Monkey, Bananas, Box\},\ l \in \{L_1, L_2, L_3\}), \\ HasBananas,\ OnBox. \end{array} \tag{4.1}$$

Actions in the domain can be denoted by symbols:

$$Walk(l),\ PushBox(l),\ ClimbOn,\ ClimbOff,\ GraspBananas. \tag{4.2}$$

There are two kinds of propositions, called "causal laws," in $\mathcal{C}$. A *static law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \tag{4.3}$$

where F and G are fluent formulas. For instance,

$$\textbf{caused } At(Bananas, l) \textbf{ if } At(Monkey, l) \wedge HasBananas \tag{4.4}$$

is a static law. The intuitive meaning of the proposition is that the location of the bananas is determined by the location of the monkey if it has the bananas. The change of the location of the bananas is an indirect effect of any action that affects the location of the monkey.

A *dynamic law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{4.5}$$

where F and G are fluent formulas and H is a formula. For instance,

$$\textbf{caused } At(Monkey, l) \textbf{ if } \top \textbf{ after } Walk(l)$$

is a dynamic law describing the effect of an action of walking.

In both propositions (4.3) and (4.5), the formula F is called the *head*. The part **if** G can be dropped if G is $\top$.

A *causal law* is a static law or a dynamic law. An *action description* is a finite set of causal laws. An action description is *definite* if the head of every causal law of it is either a literal or $\bot$.

4.1.2 Semantics

As in [Giunchiglia and Lifschitz, 1998], the semantics of $\mathcal{C}$ can be defined in terms of causal logic. An action description is mapped to a causal theory whose models are in a 1–1 correspondence with the paths in the transition system.

More precisely, any action description can be viewed as an abbreviation for a sequence of causal theories. For any action description D and any nonnegative integer m, the causal theory D_m is defined as follows. 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.

If c is a fluent, then $i : c$ means that c holds at step i, and if c is an action, then $i : c$ means that c occurs between steps i and $i + 1$.

In the description of the rules of D_m below, the following convention is used: for any formula F of the signature of D, $i{:}F$ stands for the result of prefixing all fluent symbols and action symbols in F with $i{:}$. The rules of D_m are

$$i{:}F \Leftarrow i{:}G \tag{4.6}$$

for every static law (4.3) in D and every $i \in \{0, \dots, m\}$;

$$i{+}1{:}F \Leftarrow i{+}1{:}G \ \wedge \ i{:}H \tag{4.7}$$

for every dynamic law (4.5) in D and every $i \in \{0, \dots, m-1\}$;

$$\begin{aligned} 0{:}c &\Leftarrow 0{:}c \\ 0{:}\neg c &\Leftarrow 0{:}\neg c \end{aligned} \tag{4.8}$$

for every fluent symbol c;

$$\begin{aligned} i{:}c &\Leftarrow i{:}c \\ i{:}\neg c &\Leftarrow i{:}\neg c \end{aligned} \tag{4.9}$$

for every action symbol c, and every $i \in \{0, \dots, m-1\}$.

Rules (4.8) express that the initial values of all fluents are "exogenous": they can be chosen arbitrarily. Rules (4.9) express that all actions are exogenous: whether or not an action is executed can be decided arbitrarily.

For instance, consider the following simple action description SD where there are only one fluent symbol P and only one action symbol A:

$$\begin{aligned} &\textbf{caused } P \textbf{ if } \top \textbf{ after } A \\ &\textbf{caused } P \textbf{ if } P \textbf{ after } P \\ &\textbf{caused } \neg P \textbf{ if } \neg P \textbf{ after } \neg P. \end{aligned}$$

The first line expresses that if the action A is executed, then the value of P will be caused to be true; the next two lines express the commonsense law of inertia: in the absence of any evidence to the contrary, the value of P after an event is assumed to be the same as the value before the event. This is how $\mathcal{C}$ solves the frame problem.

The causal theory SD_m for action description SD consists of

$$i+1{:}P \Leftarrow i{:}A$$
$$i+1{:}P \Leftarrow i+1{:}P \ \wedge \ i{:}P$$
$$i+1{:}\neg P \Leftarrow i+1{:}\neg P \ \wedge \ i{:}\neg P$$

for every $i \in \{0,\dots,m-1\}$ according to (4.7);

$$0{:}P \Leftarrow 0{:}P$$
$$0{:}\neg P \Leftarrow 0{:}\neg P$$

according to (4.8);

$$i{:}A \Leftarrow i{:}A$$
$$i{:}\neg A \Leftarrow i{:}\neg A$$

for every $i \in \{0,\dots,m-1\}$ according to (4.9).

It is easy to check that the models of the completion of SD_m can be written as m equivalences

$$i+1{:}P \ \equiv \ i{:}A \ \vee \ i{:}P \qquad (0 \leq i < m).$$

SD_m has 2^{m+1} models, each characterized by the truth values assigned to the constants $0:P$ and $i:A$ $(i = 0,\dots,m-1)$. For instance, one of the models of SD_2 is

$$\{\neg 0{:}P, \ \neg 0{:}A, \ \neg 1{:}P, \ 1{:}A, \ 2{:}P\}. \tag{4.10}$$

Intuitively, it means that P is false in the beginning, and remains false when action A is not executed. Then the action is executed, which will make P true.

Certain abbreviations are useful. If a is an action constant and F, G are fluent formulas, then

$$a \ \textbf{causes} \ F \ \textbf{if} \ G \tag{4.11}$$

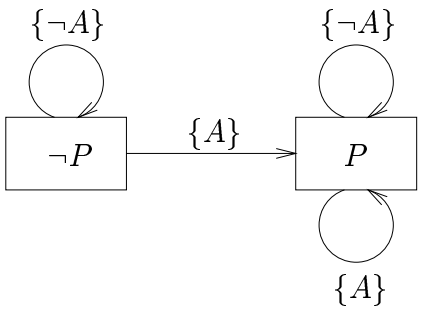

Figure 4.1: The transition system described by SD

stands for the dynamic law

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

The rule (4.11) can be used for describing a conditional effect of an action, i.e., for expressing that executing action a causes F to be true if G holds in the current state. The part **if** G can be dropped if G is $\top$. So the first line of SD can be abbreviated as

$$A \textbf{ causes } P.$$

There is also an abbreviation for the last two lines of SD:

$$\textbf{inertial } P. \tag{4.12}$$

4.1.3 States and Transitions

The models of SD_m can be visualized as paths in a "transition system"—the graph shown in Figure 4.1. The two vertices of the graph represent states; in one state, the value of the fluent P is **f**, in the other it is **t**. The edges represent transitions between states; the action a is executed in two transitions, and it is not executed in the other two.

There is a simple 1–1 correspondence between the models of SD_m and the paths of length m in this transition system. For instance, the model of SD_2 in (4.10) corresponds to the path

$$\langle \neg P,\ \neg A,\ \neg P,\ A, P \rangle.$$

Indeed, any action description describes a transition system. Consider an action description D with a set σ^{fl} of fluent symbols and a set σ^{act} of action symbols. The transition system represented by D is defined by D_0 and D_1 as we will see soon.

We can represent any interpretation of the signature of D_m in the form

$$(0{:}s_0) \cup (0{:}e_0) \cup (1{:}s_1) \cup (1{:}e_1) \cup \cdots \cup (m{:}s_m) \tag{4.13}$$

where $s_0, \ldots, s_m$ are interpretations of σ^{fl}, and $e_0, \ldots, e_{m-1}$ are interpretations of σ^{act}.

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*. One can check that according to the definitions, the graph in Figure 4.1 is indeed the transition system described by SD.

A *history* is a sequence of the form

$$\langle s_0, e_0, s_1, e_1, \ldots, s_{m-1}, e_{m-1}, s_m \rangle$$

where each $\langle s_0, e_0, s_1 \rangle, \langle s_1, e_1, s_2 \rangle, \cdots, \langle s_{m-1}, e_{m-1}, s_m \rangle$ is a transition.

Proposition [Giunchiglia and Lifschitz, 1998, Proposition 2] *For any $m > 0$, an interpretation (4.13) of the signature of D_m is a model of D_m iff*

$$\langle s_0, e_0, s_1, e_1, \ldots, s_{m-1}, e_{m-1}, s_m \rangle$$

is a history of D.

4.2 Examples

4.2.1 Monkey and Bananas

We illustrate the use of $\mathcal{C}$ by formalizing the Monkey and Bananas domain. The signature is as given in Section 4.1.1. In the following, x ranges over *Monkey*, *Bananas*, *Box*; l, l_1, l_2 range over L_1, L_2, L_3.

The first postulate expresses that there exists a location for each object at every instant:

$$\textbf{constraint } \bigvee_l At(x, l) \tag{4.14}$$

The symbol $\bigvee_l$ denotes a multiple disjunction over locations l. For a fluent formula F,

$$\textbf{constraint } F$$

stands for the static law

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

The proposition constrains the set of states: if an action description contains the proposition, every state in the corresponding transition system must satisfy F.

The second postulate expresses that each object belongs to at most one

location:

$$\textbf{caused } \neg At(x, l_1) \textbf{ if } At(x, l) \qquad (l \neq l_1). \tag{4.15}$$

The fact that an object, when moved to another location, "disappears" from its previous location can be treated as an indirect effect, or "ramification," of the moving action. This is represented by (4.15), which illustrates how $\mathcal{C}$ solves the ramification problem using static laws. Note that static laws do not mention actions, and we will soon see why the use of static laws is an attractive solution for the ramification problem.

The next group of static laws further constrains the set of states: if the monkey has the bananas, then the bananas are at the location where the monkey is; if the monkey is on the box, then the monkey is at the location where the box is.

$$\begin{array}{l} \textbf{caused } At(Bananas, l) \textbf{ if } At(Monkey, l) \wedge HasBananas \\ \textbf{caused } At(Monkey, l) \textbf{ if } At(Box, l) \wedge OnBox. \end{array} \tag{4.16}$$

The first law ensures that the change in the location of the bananas is an indirect effect of walking if the monkey has the bananas. Walking not only affects the location of the monkey, but also the location of the bananas if the monkey has them. The second effect can be described by

$$Walk(l) \textbf{ causes } At(Bananas, l) \textbf{ if } HasBananas.$$

But this law is redundant, because in the presence of the first line of (4.16), the change in the location of the bananas is an indirect effect of walking (and of any other action that affects the location of the monkey). The possibility of this simplification is what makes the postulate (4.16) attractive.

Similarly in view of the second law, the change in the location of the monkey is an indirect effect of moving the box.[1]

The effects and the preconditions of walking are described as follows:

$$\begin{array}{l} Walk(l) \textbf{ causes } At(Monkey, l) \\ \textbf{nonexecutable } Walk(l) \textbf{ if } At(Monkey, l) \\ \textbf{nonexecutable } Walk(l) \textbf{ if } OnBox. \end{array} \tag{4.17}$$

In the last two lines

$$\textbf{nonexecutable } a \textbf{ if } G \tag{4.18}$$

is an abbreviation for (4.11) when F is $\bot$. The proposition is used to represent a qualification for executing action a.

Pushing the box has two effects and three preconditions:

$$\begin{array}{l} PushBox(l) \textbf{ causes } At(Monkey, l) \\ PushBox(l) \textbf{ causes } At(Box, l) \\ \textbf{nonexecutable } PushBox(l) \textbf{ if } At(Monkey, l) \\ \textbf{nonexecutable } PushBox(l) \textbf{ if } At(Monkey, l_1) \wedge At(Box, l_2) \qquad (l_1 \neq l_2) \\ \textbf{nonexecutable } PushBox(l) \textbf{ if } OnBox. \end{array} \tag{4.19}$$

[1] Of course in this domain with only one monkey, it is not possible to move the box with the monkey on it. But if we enhance the domain to allow multiple monkeys, then this will become possible.

The descriptions of the rest of actions have a similar structure:

$ClimbOn$ **causes** $OnBox$

nonexecutable $ClimbOn$ **if** $At(Monkey, l) \wedge At(Box, l_1) \qquad (l \neq l_1)$

nonexecutable $ClimbOn$ **if** $OnBox$

$ClimbOff$ **causes** $\neg OnBox$

nonexecutable $ClimbOff$ **if** $\neg OnBox$

$GraspBananas$ **causes** $HasBananas$

nonexecutable $GraspBananas$ **if** $HasBananas$

nonexecutable $GraspBananas$ **if** $At(Monkey, l) \wedge At(Bananas, l_1) \quad (l \neq l_1)$

nonexecutable $GraspBananas$ **if** $\neg OnBox$. (4.20)

Every fluent in this domain tends to keep its previous value. The inertia rules are

$$\textbf{inertial } c \tag{4.21}$$

for every fluent symbol c from (4.1).

The concurrent execution of actions can be prohibited by postulating

$$\textbf{nonexecutable } c \wedge d \tag{4.22}$$

for every pair of distinct action symbols c, d from (4.2).

Let us call this action description MB. The planning problem given in Chapter 1 asks to find a path in the transition system described by MB that starts from the state defined by

$$At(Monkey, L_1),\ At(Bananas, L_2),\ At(Box, L_3)$$

and leads to a goal state that satisfies

$$HasBananas.$$

4.2.2 Blocks World

In the blocks world, a state is described by a set of stacks of blocks on the table.

In the following, b, b_1, b_2 range over blocks A, B, C and D; l ranges over blocks and *Table*. The symbol $On(b, l)$ denotes the fact that block b is on location l; the symbol $Move(b, l)$ denotes the action of moving block b onto location l. As in the previous example, we begin by postulating that On is a function that maps a block into a location:

$$\begin{array}{l} \textbf{constraint } \bigvee_l On(b, l) \\ \textbf{caused } \neg On(b, l_1) \textbf{ if } On(b, l) \qquad (l \neq l_1). \end{array} \tag{4.23}$$

In addition, we say that, in any state, two blocks cannot be on top of the same block at the same time:

$$\textbf{constraint } \neg(On(b, b_2) \wedge On(b_1, b_2)) \qquad (b \neq b_1). \tag{4.24}$$

The effect of moving a block is represented by the following rule:

$$Move(b, l) \textbf{ causes } On(b, l). \tag{4.25}$$

The next three postulates describe the preconditions of the action: a block can be moved only when it is clear; a block can be moved only to a position that is clear; a block cannot be moved onto a block that is being moved also:

$$\begin{array}{l} \textbf{nonexecutable } Move(b, l) \textbf{ if } On(b_1, b) \\ \textbf{nonexecutable } Move(b, b_1) \textbf{ if } On(b_2, b_1) \\ \textbf{nonexecutable } Move(b, b_1) \wedge Move(b_1, l). \end{array} \tag{4.26}$$

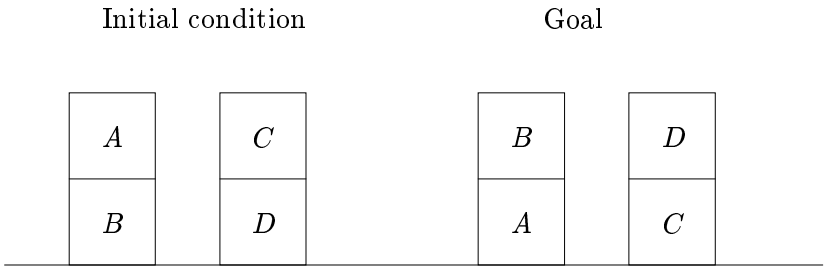

Figure 4.2: The Blocks World—A planning problem

Finally the inertia rules are

$$\textbf{inertial } On(b, l). \tag{4.27}$$

Let us call this action description BW. A typical problem in this domain is to find a sequence of moves that leads to a goal. For instance, consider the problem in Figure 4.2: given an initial configuration shown on the left, what is the shortest sequence of moves that turns it into the goal configuration shown on the right? This is a planning problem that asks to find a path in the transition system described by BW that starts from the state defined by

$$On(A, B),\ \ On(B, Table),\ \ On(C, D),\ \ On(D, Table),$$

and leads to the goal state defined by

$$On(A, Table),\ \ On(B, A),\ \ On(C, Table),\ \ On(D, C).$$

4.3 Language of the Causal Calculator

Since CCALC is an implementation of definite causal theories, it can handle definite action descriptions in $\mathcal{C}$. Indeed, all examples of $\mathcal{C}$ action descriptions we have seen

so far belong to this category.

Many commonsense reasoning problems related to $\mathcal{C}$ action descriptions can be viewed as problems of generating paths in the corresponding transition systems that satisfy certain conditions. As shown in Section 4.1.2, paths of a transition system can be obtained by computing the models of the corresponding causal theory. An action description is translated by CCALC first into a causal theory by a macro expansion mechanism and then into a set of propositional formulas using the literal completion procedure (Section 2.3). The models of the set of formulas, which correspond to paths in the transition system, are found by running SAT solvers, such as SATO [Zhang, 1997] and RELSAT [Bayardo and Schrag, 1997], in the spirit of satisfiability planning [Kautz and Selman, 1992]. Below we present how the example $\mathcal{C}$ action descriptions in the previous section can be represented in the input language of CCALC.

4.3.1 Monkey and Bananas in the Language of CCALC

A $\mathcal{C}$ input file for the Causal Calculator consists of declarations, propositions in $\mathcal{C}$ (or, more often, schemas with metavariables whose instances are propositions in $\mathcal{C}$), queries (for instance, planning problems) and comments. Among its declarations, a $\mathcal{C}$ input file usually contains a directive to include the "standard" file `C.t` which contains rewrite rules for translating from $\mathcal{C}$ into the language of causal logic, as well as various sorts, variables, constants, and domain independent causal laws that have been found to be useful in formalizing action domains.

A $\mathcal{C}$ input file for *MB* (Section 4.2.1) is shown in Figure 4.3—4.4: Figure 4.3 contains declarations for symbols used; Figure 4.4 contains the corresponding causal

```
% File: 'monkey.t'

:- include 'C.t'.

:- sorts
  thing;
  location.

:- variables
  O                                          :: thing;
  L, L1, L2                                  :: location.

:- constants
  monkey, box, bananas                       :: thing;
  l1, l2, l3                                 :: location;
  at(thing,location), onBox, hasBananas      :: inertialFluent;
  walk(location), pushBox(location),
  climbOn, climbOff, graspBananas            :: action.
```

Figure 4.3: Monkey and Bananas in the language of CCALC—Declarations

laws from Section 4.2.1.

Since CCALC is written in Prolog, the syntax of input files follows the Prolog tradition of capitalizing variables. The ranges of schematic variables declared in the variables section in Figure 4.3 are given names `thing`, `location` in the sort declaration section at the beginning. The extent of each sort is defined in the first two lines of the constant declaration section. Fluent symbols are declared `inertialFluents`: the identifier instructs CCALC to declare the symbols fluents, and moreover to postulate that the fluents are inertial, i.e., implicitly added are `inertial c` for each fluent symbol `c`. This is a built-in solution in CCALC for solving the frame problem.

The propositions in Figure 4.4 are almost identical to the causal laws from

```
constraint [\/L | at(O,L)].
caused -at(O,L1) if at(O,L) & L\=L1.

caused at(bananas,L) if hasBananas & at(monkey,L).
caused at(monkey,L) if at(box,L) & onBox.

walk(L) causes at(monkey,L).
nonexecutable walk(L) if at(monkey,L).
nonexecutable walk(L) if onBox.

pushBox(L) causes at(monkey,L).
pushBox(L) causes at(box,L).
nonexecutable pushBox(L) if at(monkey,L).
nonexecutable pushBox(L) if at(monkey,L1) & at(box,L2) & L1\=L2.
nonexecutable pushBox(L) if onBox.

climbOn causes onBox.
nonexecutable climbOn if at(monkey,L) & at(box,L1) & L\=L1.
nonexecutable climbOn if onBox.

climbOff causes -onBox.
nonexecutable climbOff if onBox.

graspBananas causes hasBananas.
nonexecutable graspBananas if hasBananas.
nonexecutable graspBananas if at(monkey,L) & at(bananas,L1) & L\=L1.
nonexecutable graspBananas if -onBox.

noconcurrency.
```

Figure 4.4: Monkey and Bananas in the language of CCALC—Causal laws

```
% File: 'monkey-test.t'

:- include 'monkey.t'.

:- plan
facts ::
 0: at(monkey,l1),
 0: at(banana,l2),
 0: at(box,l3);
goals  ::
 1..100: hasBananas.
```

Figure 4.5: Monkey and Bananas in the language of CCALC—Planning problem

Section 4.2.1. The ASCII representations of some symbols used in the language of CCALC are summarized in the following chart:

Symbol	$\neg$	$\neq$	$\wedge$	$\vee$	$\supset$	$\equiv$	$\bot$	$\top$
ASCII representation	`-`	`\=`	`&`	`++`	`->>`	`<->`	`false`	`true`

Every proposition in Figure 4.4 containing schematic variables is treated as an abbreviation for the set of $\mathcal{C}$ propositions. In a step called "grounding," CCALC replaces each variable with every object in the range of the corresponding sort; some parts of a schema turn into 0-place connectives $\top$, $\bot$. For instance, grounding turns `L\=L1` in the schema

```
caused -at(O,L1) if at(O,L) & L\=L1.
```

into $\top$ when `L` and `L1` are instantiated by different objects, and into $\bot$ otherwise.

Figure 4.5 represents the planning problem for this domain. Symbols `0:` and `1..100:` are "time stamps." `1..100:` in the goal condition instructs CCALC to

first try to find a plan of length 1, then 2, 3, and so on until it finds a solution or fails after trying length 100.

Given the query, CCALC finds a model of MB_m that satisfies the initial conditions

$$0{:}\,At(Monkey, L_1),\ 0{:}\,At(Bananas, L_2),\ 0{:}\,At(Box, L_3) \tag{4.28}$$

and the goal

$$m{:}\,HasBananas \tag{4.29}$$

where m is the smallest number for which such a model exists. CCALC takes consecutively $m = 1, 2, \ldots$ and looks for an interpretation satisfying both the completion of MB_m and formulas (4.28), (4.29). Such an interpretation will be first found for $m = 4$. It assigns the value **t** to

$$0{:}\,Walk(L_3),\ 1{:}\,PushBox(L_2),\ 2{:}\,ClimbOn,\ 3{:}\,GraspBananas.$$

Accordingly, CCALC output is as follows:

```
| ?- plan 0.

calling sato 3.1.2...
run time (seconds)                0.00
  No plan of length 1,


calling sato 3.1.2...
run time (seconds)                0.01
  No plan of length 2,


calling sato 3.1.2...
```

```
run time (seconds)                  0.01
  No plan of length 3,


calling sato 3.1.2...
run time (seconds)                  0.00

0:  at(bananas,l2)  at(box,l3)  at(monkey,l1)

ACTIONS:  walk(l3)

1:  at(bananas,l2)  at(box,l3)  at(monkey,l3)

ACTIONS:  pushBox(l2)

2:  at(bananas,l2)  at(box,l2)  at(monkey,l2)

ACTIONS:  climbOn

3:  onBox  at(bananas,l2)  at(box,l2)  at(monkey,l2)

ACTIONS:  graspBananas

4:  hasBananas  onBox  at(bananas,l2)  at(box,l2)  at(monkey,l2)


yes
```

4.3.2 Blocks World in the Language of CCalc

Figure 4.6 is a formalization of the Blocks World BW in the language of CCalc, similar to Section 4.2.2.

```
% File: 'bw.t'

:- include 'C.t'.

:- sorts
  location >> block.

:- variables
  B,B1,B2                   :: block;
  L,L1                      :: location.

:- constants
  table                     :: location;
  on(block,location)        :: inertialFluent;
  move(block,location)      :: action.

constraint [\/L | on(B,L)].
caused -on(B,L1) if on(B,L) & L\=L1.

constraint B@<B1 ->> -(on(B,B2) & on(B1,B2)).

move(B,L) causes on(B,L).

nonexecutable move(B,L) if on(B1,B).
nonexecutable move(B,B1) if on(B2,B1).
nonexecutable move(B,B1) & move(B1,L).
```

Figure 4.6: Blocks World in the language of CCALC

```
% File 'bw-test.t'.

:- include 'bw.t'.

:- constants
  a,b,c,d          :: block.

:- plan
facts::
  0: on(a,b), on(b,table), on(c,d), on(d,table);
goals::
  1..100: on(a,table), on(b,a), on(c,table), on(d,c).
```

Figure 4.7: A Blocks World planning problem

The symbol >> between the names of two sorts expresses that the second is a subsort of the first, so that every object that belongs to the second sort also belongs to the first. @< is a fixed total order between the symbols.

Figure 4.7 represents the planning problem given at the end of Section 4.2.2.

CCALC finds a model of BW_m that satisfies the initial conditions

$$0{:}\,On(A,B),\ 0{:}\,On(B,\mathit{Table}),\ 0{:}\,On(C,D),\ 0{:}\,On(D,\mathit{Table}), \tag{4.30}$$

and the goal

$$m{:}\,On(A,\mathit{Table}),\ m{:}\,On(B,A),\ m{:}\,On(C,\mathit{Table}),\ m{:}\,On(D,C). \tag{4.31}$$

where m is the smallest number for which such a model exists. CCALC takes consecutively $m = 1, 2, \ldots$ and looks for an interpretation satisfying both the completion of BW_m and formulas (4.30), (4.31). Such an interpretation will be first found for $m = 2$. The interpretation assigns the value **t** to

$$0{:}\,Move(A,\mathit{Table}),\ 0{:}\,Move(C,\mathit{Table}),\ 1{:}\,Move(B,A),\ 1{:}\,Move(D,C).$$

Note that some actions are executed concurrently.

CCALC has determined that at least two steps are needed and displayed the following solution:

```
calling sato 3.1.2...
run time (seconds)                  0.01

0:  on(a,b)  on(b,table)  on(c,d)  on(d,table)

ACTIONS:  move(a,table)  move(c,table)

1:  on(a,table)  on(b,table)  on(c,table)  on(d,table)

ACTIONS:  move(b,a)  move(d,c)

2:  on(a,table)  on(b,a)  on(c,table)  on(d,c)


yes
```

Chapter 5

New Extensions of Earlier Work

In this dissertation we show how to overcome several essential limitations of the work on causal logic, language $\mathcal{C}$ and CCALC.

5.1 Multi-valued Fluents

Most formalisms for representing properties of actions limit their attention to propositional fluents, and this is true for $\mathcal{C}$ as well. Multi-valued fluents, such as the location of an object, or the number of missionaries on a bank, can be represented in such formalisms by symbols with Boolean values, which requires introducing rules that relate these symbols to each other. For instance, in Sections 4.2.1 and 4.2.2 we described the location of an object by Boolean constants $At(x, l)$ and $On(b, l)$, and had to express the existence and the uniqueness of a location by postulates (4.14), (4.15) and (4.23). Such causal laws are needed quite often, which is inconvenient. In this respect, $\mathcal{C}$ is inferior to the language ADL (see Section 2.4) which does include symbols for multi-valued fluents.

In Section 6.1 we extend usual propositional logic by adopting a slightly more general definition of an atom that allows expressions of the form $c = v$, where v is an element of the "domain" of a symbol c. For instance, we may write $Loc(Box) = L_2$ instead of $At(Box, L_2)$. We extend causal logic and $\mathcal{C}$ in accordance with this extension.

5.2 Elaborating Actions by Attributes

Consider McCarthy's elaborations of the Missionaries and Cannibals Puzzle (Section 2.5). There is only one action, crossing, in the basic problem. We can represent this action by a symbol such as `cross(boat,bank2,1,1)` (1 missionary and 1 cannibal cross to Bank 2 using the boat). Some of McCarthy's elaborations would require that `cross` be given more arguments. In one of the elaborations (Elaboration 17), it is necessary to distinguish between rowing fast and rowing slowly, which would require an expression like `cross(boat,bank2,1,1,fast)`. In another elaboration (Elaboration 6), only one missionary and one cannibal can row, which would require to denote which of the people on the boat can row.

As McCarthy [1998] noted (Section 2.5), adding arguments to functions and predicates is what we want to avoid: if possible, we want to formalize elaborations by adding postulates. One way to achieve this goal is to distinguish between actions and "attributes." Attributes are used to elaborate the execution of actions. For instance, we may denote the action of crossing in a boat `V` by `cross(V)`. On the other hand, the destination of this action may be denoted by an attribute symbol `to(V)` whose value is a location; the number of a group `G` on a boat `V` crossing may be denoted by an attribute symbol `howmany(V,G)`; the speed of a boat `V` may be denoted by an

attribute symbol `howfast(V)`.

Such elaborations mentioned above will involve extending the formalism by adding new attribute symbols, instead of adding new arguments to the existing action symbols. This allows us to reflect elaborations by adding postulates that describe the new effects of the action in terms of the newly introduced attributes, rather than by modifying the existing description.

In Section 6.2.7 we show how attributes can be represented in an extension of $\mathcal{C}$.

5.3 Defining New Fluents

Attempts to define new fluents by causal laws in $\mathcal{C}$ often do not lead to intuitively expected results. Suppose we add to the description BW in Section 4.2.2 new fluents $Neighbor(b, b_1)$, meaning that "one of the blocks b and b_1 is on top of the other." One might be tempted to write the definition of *Neighbor* by the following causal laws:

$$\begin{array}{l} \textbf{caused } Neighbor(b, b_1) \textbf{ if } On(b, b_1) \vee On(b_1, b) \\ \textbf{caused } \neg Neighbor(b, b_1) \textbf{ if } \neg Neighbor(b, b_1). \end{array} \tag{5.1}$$

The second line of (5.1) abbreviates the set of causal laws

$$\neg i : Neighbor(b, b_1) \Leftarrow \neg i : Neighbor(b, b_1).$$

As discussed in Section 3.4, rules like this represent, intuitively, the closed-world assumption: by default, the fluent $Neighbor(b, b_1)$ is assumed to be false.

Let us call the extended description with (5.1), BW^N.

Unfortunately, the description (5.1) is not satisfactory: it does not express

that every state satisfies the condition

$$Neighbor(b, b_1) \equiv On(b, b_1) \vee On(b_1, b), \tag{5.2}$$

or equivalently, that the models of D_0 satisfy

$$0 : Neighbor(b, b_1) \equiv 0 : On(b, b_1) \vee On(b_1, b), \tag{5.3}$$

as one would intuitively expect.

To see why, consider the literal completion formula of BW_0^N for $0\!:\!Neighbor(b, b_1)$ and its negation:

$$\begin{aligned} &0\!:\!Neighbor(b, b_1) \equiv 0\!:\!Neighbor(b, b_1) \vee (On(b, b_1) \vee On(b_1, b)) \\ &0\!:\!\neg Neighbor(b, b_1) \equiv 0\!:\!\neg Neighbor(b, b_1). \end{aligned} \tag{5.4}$$

The second equivalence is a tautology, and the implication from the left to right of the first equivalence is also a tautology. Thus (5.4) is equivalent to

$$0 : On(b, b_1) \vee On(b_1, b) \supset 0 : Neighbor(b, b_1),$$

which is weaker than (5.3).

The semantics of $\mathcal{C}$ needs to be corrected to avoid such anomaly. In Section 6.2.3, we show how this can be achieved by introducing a new type of fluent constants called "statically determined."

5.4 Rigid Constants

Imagine that we want to enhance the description of the Blocks World by specifying the materials that the blocks are made of, say wood or metal, or by describing the size of the blocks. Such characteristics of blocks are not fluents because they do not

depend on the state of the system. We call them *rigid*. Rigidity can be modeled in $\mathcal{C}$ using inertial fluents: if no action is assumed to affect an inertial fluent, then its value never changes. But this treatment looks somewhat unnatural.

Modeling rigidity by fluents is also computationally inefficient. As described in Section 6.2.2, in turning an action description into a causal theory, CCALC generates atoms $i{:}c$ for fluent constants c and time stamps i. If the value of c does not change over time, then there is no need to generate copies of these atoms. This makes the size of the translation more compact, which brings computational efficiency. In Section 6.2.6, we introduce rigid constants in an extension of $\mathcal{C}$.

5.5 Defeasible Causal Laws

In the CCALC formalization of McCarthy's elaborations of the Missionaries and Cannibals Puzzle from [Lifschitz, 2000], it was necessary to make some causal laws "defeasible." For instance the formalization of the basic problem contains a proposition saying that the boat can hold two people:

$$\texttt{constraint capacity}(\texttt{boat}, 2). \tag{5.5}$$

In one of the elaborations, it was required to change this assumption: the boat can hold three people, instead of two. Rather than by removing the line above, the same effect could be obtained by adding causal laws, in the spirit of elaboration tolerance.

However, since language $\mathcal{C}$ cannot represent defeasible causal laws, that paper had to rely on causal logic directly to be able to make (5.5) defeasible. Moreover in the version of CCALC used there, only a few propositions, such as **constraint** and **nonexecutable**, could be made defeasible.

In Section 6.2.4, we illustrate how an enhancement of $\mathcal{C}$ overcomes the limitations: the semantics of defeasible causal laws can be explained in terms of the enhancement of $\mathcal{C}$; any causal law can be made defeasible. Moreover CCALC provides a convenient syntax for using defeasible causal laws.

5.6 Additive Fluents

Some action languages, including $\mathcal{C}$, allow us to talk about the effect of the concurrent execution of actions. The causal law

$$Walk(l) \textbf{ causes } At(Monkey, l)$$

is understood in $\mathcal{C}$ to imply that $At(Monkey, l)$ holds after any event that involves the execution of $Walk(l)$, even if other actions are executed concurrently.

To distinguish the events involving the concurrent execution of actions a_1 and a_2 from the events that involve a_1 but not a_2, we can write

$$a_1 \wedge a_2 \textbf{ causes } \ldots,$$
$$a_1 \wedge \neg a_2 \textbf{ causes } \ldots .$$

In some cases, unfortunately, the **causes** construct of $\mathcal{C}$ and similar languages is not directly applicable to describing the effect of the concurrent execution of actions. Consider, for instance, the effect of the action $Buy(x, n)$ (customer x buys n books) on the number of books available at a bookstore. The causal law

$$Buy(x, n) \textbf{ causes } Available(k-n) \textbf{ if } Available(k) \tag{5.6}$$

is applicable in the case when no customer other than x is buying books at the same time: $k - n$ books are available after the event if there were k books in the store

before the event. But (5.6) is not acceptable if we are interested in the concurrent execution of such actions. For instance, according to (5.6), the actions $Buy(x_1, 3)$ and $Buy(x_2, 5)$ cannot be executed concurrently, although intuitively we expect the number to be decremented by 8.

Available is an example of an "additive" fluent. An additive fluent is a fluent with numerical values such that the effect of several concurrently executed actions on it can be computed by adding the effects of the individual actions. For example, the gross receipts of a store are represented by an additive fluent: when several customers pay to different cashiers simultaneously, the gross receipts will increase by the sum of the "contributions" of the individual customers. The voltage of a battery is an additive fluent: the increase in voltage obtained by adding several cells to a battery can be computed by addition. In mechanics, the velocity of a particle is an additive fluent, because the net effect of several forces on this fluent over a time interval equals the sum of the effects of the individual forces. Additive fluents are ubiquitous; this may be the reason why adding numbers is such a useful operation.

As noted above, the effect of the concurrent execution of actions on an additive fluent is not covered by the "built-in" treatment of the concurrent execution of actions in $\mathcal{C}$. This problem was first observed in [Lifschitz, 2000] in connection with the elaborations of the Missionaries and Cannibals Puzzle that involve concurrent actions. One of the postulates adopted in that paper is that if the number of members of a group (say, missionaries) in some location (say, the left bank of the river) equals x, and a vessel arrives with y members of the group aboard, the number will become $x + y$. But this may be incorrect when several actions are executed concurrently. If, for instance, a boat is taking y missionaries to the left bank while

another boat is taking z missionaries to the right bank then the number will become $x+y-z$. To treat such examples correctly, we need to view the number of members of a group in a location as an additive fluent.

In Chapter 8 we show how the new language can be used for representing additive fluents; in Chapter 9 we apply the new version of CCALC, which can represent additive fluents and defeasible causal laws, to formalizing a few elaborations of MCP.

5.7 Nondefinite Causal Theories

As discussed in Section 3.4, it is straightforward to embed propositional logic into causal logic. The other direction, embedding causal logic into propositional logic, is more difficult. Completion gives us a partial answer: if a theory is definite, it can be turned into a propositional theory.

In Chapter 10 we show how to turn arbitrary causal theories into propositional formulas. This process includes completion as a special case. It also allows us to turn any nondefinite theory into an equivalent definite theory. Some of the theorems about causal logic can be proved more easily by turning a causal theory into an equivalent propositional theory, rather than by applying the fixpoint definition directly. In Chapter 11 we show, for instance, how the idea can be used to prove the theorem on "splitting" causal theories.

Nondefinite theories can be useful also in applications to representing commonsense knowledge. Although definite theories are widely applicable, there are cases where nondefinite theories yield more natural formalizations. An action domain of this kind, due to Marc Denecker, is discussed in [McCain, 1997, Section

7.5]:

> Imagine that there are two gears, each powered by a separate motor. There are switches that toggle the motors on and off, and there is a button that moves the gears so as to connect or disconnect them from one another. The motors turn the gears in opposite (i.e., compatible) directions. A gear is caused to turn if either its motor is on or it is connected to a gear that is turning.

A nondefinite action description representing this example in $\mathcal{C}$ is shown in Figure 5.1. The expression

default F

stands for

caused F **if** F

("There is a cause for F if F holds").

5.8 Extending CCalc

McCain's CCalc accepts $\mathcal{C}$ as an input language, but it does not handle the extended $\mathcal{C}$ presented in the next chapter which overcomes the limitations discussed here. In Section 6.5 we present the new version of CCalc that implements the extended language.

Besides the implementation of the theoretical extensions, the new CCalc provides more convenient features for compact representation. For instance, McCain's CCalc could not automatically evaluate arithmetical expressions in rules,

Notation: x ranges over $1, 2$.

Simple fluent constants:	Domains:
$MotorOn(x)$, $Turning(x)$, $Connected$	Boolean

Action constants:	Domains:
$Toggle(x)$, $Push$	Boolean

Causal Laws:

inertial $MotorOn(x)$
inertial $Connected$

$Toggle(x)$ **causes** $MotorOn(x)$ **if** $\neg MotorOn(x)$
$Toggle(x)$ **causes** $\neg MotorOn(x)$ **if** $MotorOn(x)$
$Push(x)$ **causes** $Connected$ **if** $\neg Connected$
$Push(x)$ **causes** $\neg Connected$ **if** $Connected$

caused $Turning(x)$ **if** $MotorOn(x)$
default $\neg Turning(x)$
caused $Turning(1) \equiv Turning(2)$ **if** $Connected$

Figure 5.1: Formalization of Two Gears in $\mathcal{C}$

and relied on the "`is`" predicate in underlying Prolog, so that to write a causal law such as (5.6) one had to write something like

```
buy(X,N) causes available(K1) if available(K) & K1 is K-N.
```

Another improvement is related to grounding. Rather than blindly replacing each schematic variable in causal laws with every object in the range of the corresponding sort, the new version of CCALC allows us to limit grounding to instances of the variables that satisfy a given test. We will see an example in Section 6.5.

Other new features of CCALC will be discussed in Section 6.5 also.

Chapter 6

Multi-valued Causal Logic, Action Language $\mathcal{C}+$ and CCalc 2.0

To overcome the difficulties discussed in the previous chapter, we have extended the McCain–Turner causal logic, proposed a new action language $\mathcal{C}+$ based on this extension, and redesigned and reimplemented CCalc accordingly.

6.1 Multi-valued Causal Logic

6.1.1 Multi-valued Formulas

We slightly extend formulas of the usual propositional logic to be able to represent multi-valued fluents. Differently from propositional logic, where each symbol is mapped to either **f** or **t**, in "multi-valued" propositional logic defined in this section,

a symbol can be mapped to an element of a certain finite set of values.

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. 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 $v \in Dom(c)$. A *formula* of σ is a propositional combination of atoms.

For instance, the following atoms may be used to describe the location of an agent in an apartment:

$$Loc = Kitchen,\ Loc = LivingRoom,\ Loc = Bathroom,\ Loc = Bedroom \tag{6.1}$$

where Loc is a constant with the domain

$$\{Kitchen, LivingRoom, Bathroom, Bedroom\}.$$

An *interpretation* of σ is a function that maps every element of σ to an element of its domain. An interpretation I *satisfies* an atom $c{=}v$ (symbolically, $I \models c{=}v$) if $I(c) = v$. For instance, the fact that the agent is in the kitchen can be described by an interpretation satisfying the first of the atoms in (6.1), so that the others are not satisfied.

The satisfaction relation is extended from atoms to arbitrary formulas according to the usual truth tables for the propositional connectives.

The following definitions are standard in logic. Two formulas or sets of formulas are *equivalent* to each other if they are satisfied by the same interpretations. A *model* of a set Γ of formulas is an interpretation that satisfies all formulas in Γ. If Γ has a model, it is said to be *consistent*, or *satisfiable*. If every model of Γ satisfies a formula F then we say that Γ *entails* F and write $\Gamma \models F$.

A *Boolean* constant is one whose domain is the set of truth values $\{\mathbf{f}, \mathbf{t}\}$. A *Boolean* signature is one whose constants are Boolean. If c is a Boolean constant, we will sometimes use c as shorthand for the atom $c = \mathbf{t}$. When the syntax and the semantics defined above are restricted to Boolean signatures and to formulas that do not contain $\mathbf{f}$, they turn into the usual syntax and semantics of classical propositional formulas. In principle, the domain of a constant can be a singleton.

Recall that, according to the definition, an atom is an equality whose left-hand side is a constant c, and whose right-hand side is an element of the domain of c. An expression of the form $c = d$, where both c and d are constants, will be understood as an abbreviation for the disjunction

$$\bigvee_{v \in Dom(c) \cap Dom(d)} (c = v \wedge d = v).$$

The symbol $\neq$ will be used to abbreviate the negation of an equality of either kind.

6.1.2 Multi-valued Causal Logic

By a *(multi-valued) causal rule* we mean an expression of the form $F \Leftarrow G$ ("F is caused if G is true"), where F and G are multi-valued formulas of a given signature σ. A *(multi-valued) causal theory* is a finite set of causal rules. From now on, we will often drop the word "multi-valued."

As in the Boolean case, 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.

For example, take

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

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

$$c{=}1 \Leftarrow c{=}1. \tag{6.2}$$

The interpretation I defined by $I(c) = 1$ is a model of T. Indeed,

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

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

It follows that causal theory (6.2) entails $c{=}1$.

Consider now what happens if we add the rule

$$c{=}2 \Leftarrow \top \tag{6.3}$$

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

The extended theory does not entail $c = 1$; it entails $c = 2$. This example illustrates the nonmonotonicity of the logic. Intuitively, rule (6.2) expresses that 1 is "the default value" of c, and rule (6.3) overrides this default.

If the rule

$$c{=}2 \Leftarrow c{=}2 \tag{6.4}$$

is added to (6.2) instead of (6.3), we will get a causal theory with two models. This theory entails $c{=}1 \vee c{=}2$.

6.1.3 Multi-valued Completion

As in the McCain–Turner causal logic, a causal theory is *definite* if the head of every rule of it is an atom or $\bot$. For instance, causal theory (6.2) is definite. Causal theory (3.4) from Section 3.3 is, strictly speaking, not definite, but it can be turned into a definite theory by replacing $\neg q$ in the head of the last rule with the equivalent atom:

$$\begin{aligned} p &\Leftarrow q, \\ q &\Leftarrow q, \\ q{=}\mathbf{f} &\Leftarrow \neg q. \end{aligned} \tag{6.5}$$

The "multi-valued completion" process described below extends the literal completion for the McCain–Turner causal theories. It reduces the problem of finding a model of a definite causal theory to the problem of finding a model of a set of formulas.

Take a definite causal theory T of a signature σ. We say that an atom $c{=}v$ of σ is *trivial* if the domain of c is a singleton. For each nontrivial atom A, the *completion formula* for A is the formula

$$A \equiv G_1 \vee \cdots \vee G_n$$

where $G_1, \ldots, G_n$ $(n \geq 0)$ are the bodies of the rules of T with head A. The *(multi-valued) completion* of T is obtained by taking the completion formulas for all nontrivial atoms of σ, along with the formula $\neg F$ for each constraint $\bot \Leftarrow F$ in T.

As in the McCain–Turner causal logic, the following proposition holds.

Proposition 1 *The models of a definite causal theory are precisely the models of its completion.*

For instance, the completion of (6.2) is

$$\begin{aligned} &c=1 \equiv c=1, \\ &c=v \equiv \bot \qquad (v \in Dom(c) \setminus \{1\}) \end{aligned} \tag{6.6}$$

if $|Dom(c)| > 1$. Otherwise the atom $c=1$ is trivial, and the completion is empty. In both cases, the only model of the completion is defined by $I(c) = 1$. As discussed in Section 6.1.2, this is the only model of (6.2).

After adding rule (6.3), the completion turns into

$$\begin{aligned} &c=1 \equiv c=1, \\ &c=2 \equiv \top, \\ &c=v \equiv \bot \qquad (v \in Dom(c) \setminus \{1,2\}). \end{aligned}$$

The only model of these formulas is defined by $I(c) = 2$.

The completion of (6.5) is

$$\begin{aligned} p &\equiv q, \\ p=\mathbf{f} &\equiv \bot, \\ q &\equiv q, \\ q=\mathbf{f} &\equiv \neg q, \end{aligned} \tag{6.7}$$

which corresponds to (3.5).

The assertion of Proposition 1 would be incorrect if we did not restrict the completion process to nontrivial atoms. Consider, for instance, the causal theory whose signature consists of one constant c with the domain $\{0\}$, and whose set of rules is empty. If the process of completion were extended to trivial atoms then the completion of this theory would be $c=0 \equiv \bot$, which is inconsistent.

6.2 Action Language $\mathcal{C}+$

6.2.1 Syntax of $\mathcal{C}+$

Constants in $\mathcal{C}+$ are divided into two groups: *fluent constants* and *action constants*. Furthermore, the fluent constants are assumed to be partitioned into *simple* and *statically determined*. By a *fluent formula* we mean 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.

A *static law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \tag{6.8}$$

where F and G are fluent formulas. An *action dynamic law* is an expression of the form (6.8) in which F is an action formula and G is a formula. A *fluent dynamic law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{6.9}$$

where F and G are fluent formulas and H is a formula, provided that F does not contain statically determined constants. A *causal law* is a static law, an action dynamic law, or a fluent dynamic law. An *action description* is a finite set of causal laws.

An action description D is *definite* if the head of every causal law of D is an atom or $\bot$.

6.2.2 Semantics of $\mathcal{C}+$

Just as the semantics of $\mathcal{C}$ is defined in terms of the McCain–Turner causal logic, the semantics of $\mathcal{C}+$ can be defined in terms of multi-valued causal logic.

For any action description D and any nonnegative integer m, the causal theory D_m is defined as follows. As in $\mathcal{C}$, the signature of D_m consists of the pairs $i{:}c$ such that

- $i \in \{0, \dots, m\}$ and c is a fluent constant of D, or
- $i \in \{0, \dots, m-1\}$ and c is an action constant of D.

The domain of $i{:}c$ is the same as the domain of c.

The rules of D_m are

$$i{:}F \Leftarrow i{:}G \tag{6.10}$$

for every static law (6.8) in D and every $i \in \{0, \dots, m\}$, and for every action dynamic law (6.8) in D and every $i \in \{0, \dots, m-1\}$;

$$i{+}1{:}F \Leftarrow (i{+}1{:}G) \wedge (i{:}H) \tag{6.11}$$

for every fluent dynamic law (6.9) in D and every $i \in \{0, \dots, m-1\}$;

$$0{:}c{=}v \Leftarrow 0{:}c{=}v \tag{6.12}$$

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 (6.12) are included only when c is simple, so that the initial values of statically determined fluents are not assumed to be exogenous (see Section 4.1.2). We will see in the next section why this is useful.

Similarly, the assumption (4.9) that the execution of an action is exogenous is not built into the semantics of $\mathcal{C}+$, so that we need to write it explicitly if an

action is exogenous. We will see later in Section 6.2.4 and Section 8.3 when it is necessary to lift the exogeneity assumption for actions.

The definitions of states, transitions, histories are the same as in $\mathcal{C}$ (Section 4.1.3).

Proposition 2 *For any transition $\langle s, e, s' \rangle$, s and s' are states.*

We identify an interpretation I in the sense of Section 6.1.1 with the set of atoms that are satisfied by this interpretation, that is to say, with the set of atoms of the form $c = I(c)$. This convention allows us to represent any interpretation of the signature of D_m in the form

$$(0{:}s_0) \cup (0{:}e_0) \cup (1{:}s_1) \cup (1{:}e_1) \cup \cdots \cup (m{:}s_m) \tag{6.13}$$

where $s_0, \ldots, s_m$ are interpretations of σ^{fl}, and $e_1, \ldots, e_{m-1}$ are interpretations of σ^{act}.

Proposition 3 *For any $m > 0$, an interpretation (6.13) of the signature of D_m is a model of D_m iff*

$$\langle s_0, e_0, s_1, e_1, \ldots, s_{m-1}, e_{m-1}, s_m \rangle$$

is a history of D.

6.2.3 Statically Determined Fluents

The problem with defined fluents discussed in Section 5.3 can be corrected by classifying these fluents as statically determined. For instance, in the extended Blocks World domain BW^N in Section 5.3, if fluents $Neighbor(b, b_1)$ are declared statically determined, then the extent of the *Neighbor* relation is defined by the equation (5.2), as desired. To see this, note that the completion formulas of BW_0^N for

$0\colon Neighbor(b, b_1)$ and its negation are now

$$\begin{aligned} 0\colon Neighbor(b, b_1) &\equiv 0\colon On(b, b_1) \vee On(b_1, b) \\ 0\colon \neg Neighbor(b, b_1) &\equiv 0\colon \neg Neighbor(b, b_1). \end{aligned} \tag{6.14}$$

The second equivalence is a tautology, and the first equivalence is exactly (5.3).

The transition system described by BW^N is isomorphic to the one described by BW: every state of the latter can be turned into the corresponding state of the former by assigning to $Neighbor(b, b_1)$ the truth values defined by (5.2).

The following theorem describes a general method of defining fluents in $\mathcal{C}+$.

Proposition 4 *Let D be an action description whose signature is σ, Q a set of statically determined fluent constants such that $\sigma \cap Q = \emptyset$, and D_Q an action description which consists of causal laws of the form*

$$\textbf{caused } q \textbf{ if } F$$

where $q \in Q$ and F is a formula of σ, and the causal laws

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

for all $q \in Q$. Then the transition system of $D \cup D_Q$ is isomorphic to the transition system of D.

6.2.4 Defeasible Causal Laws

Using statically determined fluents, any static law can be made defeasible. A defeasible static law has the form

$$\textbf{caused } F \textbf{ if } G \textbf{ unless } ab \tag{6.15}$$

where ab is a statically determined fluent constant. It stands for the pair of causal laws

$$\begin{array}{l} \textbf{caused } F \textbf{ if } G \wedge \neg ab \\ \textbf{default } \neg ab \end{array} \tag{6.16}$$

(Recall that the second law stands for **caused** $\neg ab$ **if** $\neg ab$). Under exceptional circumstances where ab is true, causal law (6.15) becomes ineffective.

For instance, a defeasible form of proposition (5.5) can be represented in the new language by

$$\textbf{constraint } \mathit{Capacity}(\mathit{Boat}) = 2 \textbf{ unless } \mathit{AbBoat}.$$

Similarly, any action dynamic law can be made defeasible: an action dynamic law (6.15) where ab is a Boolean action constant stands for the pair of causal laws (6.16).

A defeasible fluent dynamic law has the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \textbf{ unless } ab \tag{6.17}$$

where ab is a Boolean action constant. It stands for the pair of causal laws

$$\begin{array}{l} \textbf{caused } F \textbf{ if } G \textbf{ after } H \wedge \neg ab \\ \textbf{default } \neg ab. \end{array} \tag{6.18}$$

Under exceptional circumstances where ab is true, causal law (6.17) becomes ineffective.

For instance, in the Monkey and Bananas domain,

$$\mathit{PushBox}(l) \textbf{ causes } \mathit{At}(\mathit{Box}, l) \textbf{ unless } \mathit{AbBox} \tag{6.19}$$

expresses that pushing the box normally involves changing the location of the box. Suppose we want to enhance the description by postulating that the box is not movable if it is too big. This can be done by adding

$$\textbf{caused } \mathit{AbBox} \textbf{ if } \mathit{BigBox}. \tag{6.20}$$

If the box is too big, then *AbBox* is caused and this makes (6.19) ineffective. On the other hand, intuitively, when there are no exceptions, the **unless** clause in (6.19) can be disregarded. The following proposition makes the claim precise:

Proposition 5 (*a*) *Let D be an action description, L a static causal law, and ab a Boolean statically determined fluent. If ab does not occur in the heads of any causal laws of D, then the transition system described by $D \cup \{L \textbf{ unless } ab\}$ is exactly the transition system described by $D \cup \{L\} \cup \{\textbf{caused } \neg ab\}$.*

(*b*) *Let D be an action description, L a dynamic causal law, and ab a Boolean action constant. If ab does not occur in the heads of any causal laws of D, then the transition system described by $D \cup \{L \textbf{ unless } ab\}$ is exactly the transition system described by $D \cup \{L\} \cup \{\textbf{caused } \neg ab \textbf{ after } \top\}$.*

Notice that this method of making causal laws defeasible was not possible in $\mathcal{C}$: statically determined fluents were not available and action constant *ab* could not be made non-exogenous due to the built-in exogeneity assumption (4.9) for all actions.

6.2.5 Solving the Qualification Problem in $\mathcal{C}+$

The qualification problem is the problem of representing properties of actions in a way that makes new conditions for the successful performance of an action express-

ible by adding new propositions. This is a special case of the problem of elaboration tolerance.

We can distinguish between two kinds of conditions for the successful performance of an action [Reiter, 1991]. It may happen that the action is simply not executable when the condition is violated. Or it may happen that some of the usual effects of the action do not hold in the resulting state even if the action was executed. Accordingly, we can distinguish between two parts of the qualification problem—one deals with executability qualifications, and the other with effect qualifications.

We can further distinguish between two kinds of executability qualifications—those stated explicitly, in terms of preconditions, and those expressed implicitly by constraints on the states. For instance, the fact that the monkey cannot walk if it is on the box can be expressed explicitly by adding **nonexecutable** proposition as in (4.16) (Section 4.2.1); the fact that one cannot buy more books than those available in the bookstore (Section 5.6) is expressed implicitly by the assumption that *Available* has nonnegative values. Executability qualifications can be represented in both $\mathcal{C}$ and $\mathcal{C}+$.

On the other hand, effect qualifications can be expressed in $\mathcal{C}+$, but not in $\mathcal{C}$; the $\mathcal{C}+$ solution involves the use of defeasible causal laws which $\mathcal{C}$ cannot represent. As in the previous section, the fact that a very big box accounts for an exception to the usual effect of pushing action can be represented by the combination of causal law (6.19) that allows exceptions and causal law (6.20) that specifies an exception.

6.2.6 Rigid Constants

A fluent constant c in the signature of an action description D is *rigid* (relative to D) if, for every transition $\langle s, e, s' \rangle$ in the transition system represented by D, $s'(c) = s(c)$. Intuitively, rigid constants represent the fluents whose values are not affected by any events.

The expression

$$\textbf{rigid } c$$

stands for the set of causal laws

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

for all $v \in Dom(c)$. It is clear that c is rigid relative to any action description containing these laws.

As noted in Section 5.4, one of the reasons why rigid constants are interesting is that, under some conditions, their presence allows us to make the causal theories D_m more compact, which can be computationally advantageous. Let R be a set of fluent constants that are rigid relative to D. Denote by D_m^R the causal theory whose constants and causal rules are obtained from the constants and causal rules of D_m by dropping the time stamps before each constant from R. For any interpretation I of the signature of D_m, by I^R we denote the interpretation of the signature of D_m^R defined by the formulas

$$I^R(c) = I(0:c) \qquad \text{if } c \in R,$$
$$I^R(i{:}c) = I(i{:}c) \qquad \text{if } c \notin R.$$

Proposition 6 *If*

(i) every constant in R is statically determined, and

(ii) for every causal law in D that contains a constant from R in the head, all constants occurring in this law belong to R,

then the mapping $I \mapsto I^R$ is a 1–1 correspondence between the models of D_m and the models of D_m^R.

Thus dropping the time stamps in front of the rigid constants from R does not affect the meaning of D_m if, first, R contains no simple constants, and second, no constant from R "causally depends" on a constant that does not belong to R.

The following example shows that the assertion of Proposition 6 would be incorrect without the first condition. Take D to be

rigid p

default p

where p is a Boolean simple fluent, and let $R = \{p\}$. Then D_1 is

$$\begin{aligned}
\bot &\Leftarrow (1{:}p) \wedge \neg(0{:}p) \\
\bot &\Leftarrow \neg(1{:}p) \wedge (0{:}p) \\
0{:}p &\Leftarrow 0{:}p \\
1{:}p &\Leftarrow 1{:}p \\
0{:}\neg p &\Leftarrow 0{:}\neg p
\end{aligned}$$

and D_1^R is

$$\begin{aligned}
\bot &\Leftarrow p \wedge \neg p \\
\bot &\Leftarrow \neg p \wedge p \\
p &\Leftarrow p \\
\neg p &\Leftarrow \neg p.
\end{aligned}$$

The interpretation $\{p = \mathbf{f}\}$ is a model of D_1^R, but it does not have the form I^R for any model I of D_1.

The following example shows that the assertion of Proposition 6 would be incorrect without the second condition. Take D to be

$$\textbf{caused } p \textbf{ if } q$$

$$\textbf{exogenous } q,$$

where p and q are statically determined fluent constants, and let $R = \{p\}$. Then D_1 is

$$\begin{aligned}
0{:}p &\Leftarrow 0{:}q \\
1{:}p &\Leftarrow 1{:}q \\
0{:}q &\Leftarrow 0{:}q \\
1{:}q &\Leftarrow 1{:}q \\
\neg 0{:}q &\Leftarrow \neg 0{:}q \\
\neg 1{:}q &\Leftarrow \neg 1{:}q
\end{aligned}$$

and D_1^R is

$$\begin{aligned}
p &\Leftarrow 0{:}q \\
p &\Leftarrow 1{:}q \\
0{:}q &\Leftarrow 0{:}q \\
1{:}q &\Leftarrow 1{:}q \\
\neg 0{:}q &\Leftarrow \neg 0{:}q \\
\neg 1{:}q &\Leftarrow \neg 1{:}q.
\end{aligned}$$

The interpretation $\{p = \mathbf{t}, 0{:}q = \mathbf{f}, 1{:}q = \mathbf{t}\}$ is a model of D_1^R, but it does not have the form I^R for any model I of D_1.

6.2.7 Action Attributes

Syntactically an attribute is a non-Boolean exogenous action constant. The domain of every attribute of an action includes the special value *None*, which the attribute

is required to take if and only if the action is not executed. For this purpose we postulate

$$\textbf{caused } \bot \textbf{ if } \top \textbf{ after } (attr = None) \equiv a \tag{6.21}$$

for every attribute *attr* of action a.

An expression of the form

$$\textbf{always } F$$

in $\mathcal{C}+$ stands for

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

Thus (6.21) can be abbreviated as

$$\textbf{always } (attr = None) \equiv a.$$

Note that this treatment of attributes was not possible in $\mathcal{C}$, since every action constant in $\mathcal{C}$ was Boolean.

6.3 Comparison with ADL

To clarify the relation of $\mathcal{C}+$ to the language ADL mentioned in Section 2.4, we show how Pednault's idea of "update conditions" can be incorporated into the syntactic framework of Section 6.2.1.

As a preliminary step, consider a multi-valued propositional signature σ whose constants have the same domain *Dom*. The elements of *Dom* will be called *values*. The concept of a formula of a signature σ can be extended as follows. A *term* is a constant of σ, a value, or a variable (from a fixed infinitely countable set). An *extended atom* is an expression of the form $t = v$ where t is a term and

v is a value. *Extended formulas* are formed from atoms using propositional connectives and quantifiers, as in first-order logic. We will sometimes identify a closed extended formula F with the formula in the sense of Section 6.1.1 that is obtained from F as follows: first, eliminate from F all quantifiers by replacing each subformula of the form $\forall x G(x)$ with $\bigwedge_v G(v)$, where v ranges over Dom, and each $\exists x G(x)$ with $\bigvee_v G(v)$; second, replace all occurrences of atoms of the form $v = v$ with $\top$, and all occurrences of atoms of the form $v = w$, where v is a value different from w, with $\bot$. This convention allows us, for instance, to talk about the satisfaction relation between interpretations of σ and closed extended formulas.

Consider a multi-valued signature σ partitioned into *fluent constants* σ^{fl} and *action constants* σ^{act}, such that all fluent constants have the same domain, and all action constants are Boolean. An *ADL action description* consists of

- a closed extended formula $Precond^a$ of signature σ^{fl} for every action constant a, and
- an extended formula $Update^a_c(x)$ of signature σ^{fl}, with no free variables other than x, for every action constant a and every fluent constant c.

An ADL action description is *consistent* if, for every action constant a, every fluent constant c, and every pair of distinct values v and w, formula $Precond^a$ entails

$$\neg(Update^a_c(v) \wedge Update^a_c(w)).$$

Let D be a consistent ADL action description, s and s' interpretations of σ^{fl}, and a an action constant. We say that s' is the *result of executing a in s according to D* if

$$s \models Precond^a$$

and, for every fluent constant c,

$$s'(c) = \begin{cases} v & \text{if } s \models \mathit{Update}_c^a(v), \\ s(c) & \text{if } s \models \neg\exists x \mathit{Update}_c^a(x). \end{cases}$$

Now we will define a translation from this version of ADL into $\mathcal{C}+$. In the $\mathcal{C}+$ counterpart of an ADL action description D, all fluent constants of D are treated as simple. The propositions of this $\mathcal{C}+$ action description are

$$\begin{array}{l} \textbf{inertial } c \\ \textbf{exogenous } a \\ \textbf{nonexecutable } a \textbf{ if } \neg \mathit{Precond}^a \\ a \textbf{ causes } c{=}v \textbf{ if } \mathit{Update}_c^a(v) \end{array} \tag{6.22}$$

for every fluent constant c, action constant a, and value v.

In the following theorem, we identify a Boolean action constant e with the event that maps e to **t** and maps every other action constant to **f**.

Theorem 1 *For any consistent ADL action description D, any interpretations s, s' of σ^{fl}, and any $e \in \sigma^{act}$, s' is the result of executing e in s according to D iff transition $\langle s, e, s' \rangle$ is a transition of the counterpart of D in language $\mathcal{C}+$.*

The version of ADL described above is significantly less expressive than $\mathcal{C}+$. ADL is mapped here into the subset of $\mathcal{C}+$ that includes no statically determined fluent constants; it has neither concurrent actions nor non-inertial fluents; there are no static laws or action dynamic laws in it, and consequently it does not solve the ramification problem.

6.4 Eliminating Multi-valued Constants

In fact, the extension of causal logic by multi-valued constants described in this chapter is not essential in the sense that multi-valued constants can be eliminated in favor of Boolean constants: we can replace a multi-valued constant c with a family of Boolean constants, one for each element of $Dom(c)$. In this section we show how this idea applies to multi-valued formulas and then extend it to multi-valued causal theories and to $\mathcal{C}+$.

6.4.1 Eliminating Multi-valued Constants from Formulas

Begin with a multi-valued propositional signature σ, and a constant $c \in \sigma$. By σ_c we denote the signature obtained from σ by replacing constant c with Boolean constants $c(v)$ for all $v \in Dom(c)$.

Let Γ be a set of formulas of signature σ, and Γ' a set of formulas of signature σ_c. We will say that Γ' *correctly reduces* c in Γ (to a family of Boolean constants) if the following holds: an interpretation of σ_c is a model of Γ' iff it corresponds to a model of Γ.

Let $elim_c$ be the formula

$$\bigvee_v c(v) = \mathbf{t} \ \wedge \bigwedge_{v \neq v'} (c(v) = \mathbf{f} \vee c(v') = \mathbf{f}) \,. \tag{6.23}$$

Notice that the models of $elim_c$ are precisely the interpretations of σ_c that correspond to an interpretation of σ.

For any formula F of σ, by F_c we denote the formula obtained from F by replacing each occurrence of an atom $c = v$ with $c(v) = \mathbf{t}$. The *elimination* of c from Γ is the set of formulas $\{F_c : F \in \Gamma\} \cup \{elim_c\}$.

Proposition 7 *For any set* Γ *of formulas and any constant* c*, the elimination of* c *from* Γ *correctly reduces* c *in* Γ*.*

6.4.2 Eliminating Multi-valued Constants from Causal Theories

Begin with a causal theory T whose signature is σ, and a constant $c \in \sigma$. We understand the notation σ_c as in the previous section. We will say that a causal theory T' with signature σ_c *correctly reduces* c in T (to a family of Boolean constants) if the following holds: an interpretation of σ_c is a model of T' iff it corresponds to a model of T.

General Elimination Method for Causal Theories

The *general elimination* of c from T is the causal theory with signature σ_c obtained by replacing each occurrence of an atom $c = v$ in T with $c(v) = \mathbf{t}$, and adding the causal rule

$$elim_c \Leftarrow \top. \tag{6.24}$$

Proposition 8 *For any causal theory* T *and any constant* c*, the general elimination of* c *from* T *correctly reduces* c *in* T*.*

A drawback of this simple elimination method is that rule (6.24) is not definite. For this reason, even in application to a definite theory, this process leads to a theory that is not definite. Since definiteness is useful, we next introduce another elimination method that preserves it.

Definite Elimination Method for Causal Theories

The *definite elimination* of c from T is the causal theory with signature σ_c obtained by replacing each occurrence of an atom $c=v$ in T with $c(v)=\mathbf{t}$, and adding the causal rules

$$c(v')=\mathbf{f} \Leftarrow c(v)=\mathbf{t} \tag{6.25}$$

for all $v, v' \in Dom(c)$ such that $v \neq v'$, and also adding

$$\bot \Leftarrow \bigwedge_{v} c(v)=\mathbf{f}. \tag{6.26}$$

Proposition 9 *For any causal theory T and any constant c such that (i) $Dom(c)$ has at least two elements, and (ii) every head in which c occurs is an atom, the definite elimination of c from T correctly reduces c in T.*

6.4.3 Eliminating Multi-valued Constants from $\mathcal{C}+$

A multi-valued constant in an action description can be replaced by a family of Boolean constants using methods similar to those introduced for causal theories.

We will say that an action description D' with signature σ_c *correctly reduces* c in D (to a family of Boolean fluent constants) if the following holds.

- s is a state of D iff s' is a state of D'.
- $\langle s, e, s_1 \rangle$ is a transition of D iff $\langle s', e', s'_1 \rangle$ is a transition of D'.

General Elimination Method for Action Descriptions

The *general elimination* of a fluent or action constant c from D is the action description with the signature σ_c obtained by replacing each occurrence of an atom $c=v$

in D with $c(v)=\mathbf{t}$, and adding the static law

$$\textbf{caused } elim_c \textbf{ if } \top. \tag{6.27}$$

Proposition 10 *For any action description D and any constant c, the general elimination of c from D correctly reduces c in D.*

Definite Elimination Method for Action Descriptions

The *definite elimination* of a constant c from D is the action description with action symbols σ_c^{act} and fluent symbols σ_c^{fl} obtained by replacing each occurrence of an atom $c=v$ in D with $c(v)=\mathbf{t}$, and adding the causal laws

$$\textbf{caused } c(v')=\mathbf{f} \textbf{ if } c(v)=\mathbf{t} \tag{6.28}$$

for all $v, v' \in Dom(c)$ such that $v \neq v'$, and also adding the causal law

$$\textbf{caused } \bot \textbf{ if } \bigwedge_v c(v)=\mathbf{f}. \tag{6.29}$$

Proposition 11 *For any action description D and any constant c such that (i) $Dom(c)$ has at least two elements, and (ii) any head in which c occurs is an atom, the definite elimination of c from D correctly reduces c in D.*

6.5 CCALC 2.0

We have redesigned and reimplemented CCALC according to the extensions of the causal logic and $\mathcal{C}+$ described above. The new CCALC is available at

`http://www.cs.utexas.edu/users/tag/ccalc/` .

The input language of the new CCALC provides a convenient and concise syntax for describing action descriptions in $\mathcal{C}+$. The new version of CCALC has been successfully applied to challenging problems in the theory of commonsense knowledge [Campbell and Lifschitz, 2003], [Akman *et al.*, 2004] and to the formalization of multi-agent computational systems [Artikis *et al.*, 2003a; Artikis *et al.*, 2003b; Chopra and Singh, 2003].

Compare Figures 4.3—4.5 with Figures 6.1—6.3, a description of the monkey and bananas domain in the language of the new CCALC.

Constant declarations are now divided into two parts: object declarations and constant declarations. Object declarations define the extents of sorts; constant declarations define fluent and action symbols, along with the values to which the symbols can be mapped. The set of values is specified in parentheses, as in the expression

```
inertialFluent(location).
```

The declaration

```
onBox,hasBananas        :: inertialFluent
```

is understood as shorthand for

```
onBox,hasBananas        :: inertialFluent(Boolean)
```

Notice that we declare the actions `exogenousAction` to distinguish them from non-exogenous actions. Upon reading the declaration

```
c :: exogenousAction,
```

CCALC adds

exogenous c

```
% File: 'monkey'

:- sorts
  thing;
  location.

:- objects
  monkey,bananas,box     :: thing;
  l1,l2,l3               :: location.

:- variables
  L                      :: location.

:- constants
  loc(thing)             :: inertialFluent(location);
  onBox,hasBananas       :: inertialFluent;

  walk(location),
  pushBox(location),
  climbOn,
  climbOff,
  graspBananas           :: exogenousAction.
```

Figure 6.1: Monkey and Bananas in the language of the new CCALC—Declarations

to the action description automatically.

Since we can represent the location of an object using location-valued fluent symbols, we do not need to write rules such as (4.14) and (4.15).

The line

```
pushBox(L) causes walk(L)
```

is the action dynamic law that stands for

```
caused walk(L) if pushBox(L).
```

```
walk(L) causes loc(monkey)=L.
nonexecutable walk(L) if loc(monkey)=L.
nonexecutable walk(L) if onBox.

pushBox(L) causes loc(box)=L.
pushBox(L) causes walk(L).
nonexecutable pushBox(L) if loc(monkey)\=loc(box).

climbOn causes onBox.
nonexecutable climbOn if loc(M)\=loc(box).
nonexecutable climbOn if onBox.

climbOff causes -onBox.
nonexecutable climbOff if -onBox.

graspBananas causes hasBananas.
nonexecutable graspBananas if hasBananas.
nonexecutable graspBananas if loc(monkey)\=loc(bananas).
nonexecutable graspBananas if -onBox.

noconcurrency.
```

Figure 6.2: Monkey and Bananas in the language of the new CCALC—Causal laws

```
:- query
maxstep :: 2..4;
0: loc(alice)=l1,
   loc(bananas)=l2,
   loc(box)=l3;
maxstep: hasBananas(alice).
```

Figure 6.3: Monkey and Bananas in the language of the new CCALC—Planning problem

Due to this law saying that pushing the box involves walking, we do not need to repeat the description of the effects and preconditions of pushing which are also the effects and preconditions of walking.

The expression

```
loc(monkey)\=loc(box)
```

(Recall that `\=` is the ASCII representation of $\neq$) is shorthand for

```
-[\/L | loc(monkey)=L & loc(box)=L].
```

Figure 6.3 is a counterpart of Figure 4.5 in the language of the new CCALC. A query consists of two components. One is an integer, called `maxstep`, that specifies the length of the paths the query is about. It is similar to the maximum time implicitly specified in the goal condition of Figure 4.5; its value determines how to turn the given action description into a sequence of causal theories. The second component is a set of formulas constraining the paths of interest.

Figure 6.4 is a description of the blocks world in the language of the new CCALC. Function constant `loc` represents an operation that turns a block into a fluent. This fluent is inertial, and its value is a location. The operation denoted

by `move` turns a block into an action—moving that block. Operation `destination` gives an attribute of that action—the destination of the move. Upon processing the declaration of an attribute *attr*, CCALC automatically includes (6.21).

In the last causal law of Figure 6.4, a "where" clause, containing a test, is appended. The clause instructs CCALC to limit grounding to instances of the schematic variables that satisfy the given test, which produces less number of grounded instances: in all instances of the last causal law that CCALC generates, `B` and `B1` are different from each other.

Figure 6.5 shows a query on the domain description in Figure 6.4, a counterpart for the query from Figure 4.7.

Figure 6.6 is an extension of Figure 6.4. The word `sdFluent` in the declaration of

`neighbor(block,block)` stands for "statically determined fluent constant." Recall that '`++`' is the ASCII representation of '$\vee$' in the language of CCALC.

More features of the language of the new CCALC will be presented in Section 7.3.

6.6 Proving the Unsolvability of Planning Problems in CCALC

For a planning problem described by a query with its maximum number of steps specified, CCALC can find, in principle, a plan of that length if such a plan exists; if it determines that a plan of the given length does not exist, it answers no. However, such queries cannot help us establish that a problem cannot be solved in any number

```
% File: 'bw'

:- sorts
  location >> block.

:- objects
  table                     :: location.

:- constants
  loc(block)                :: inertialFluent(location);
  move(block)               :: action;
  destination(block)        :: attribute(location) of move(block).

:- variables
  B,B1                      :: block;
  L                         :: location.

% effect of moving a block
move(B) causes loc(B)=L if destination(B)=L.

% a block can be moved only when it is clear
nonexecutable move(B) if loc(B1)=B.

% a block can be moved only to a position that is clear
nonexecutable move(B)
   if destination(B)=loc(B1) & destination(B)\=table.

% a block can't be moved onto a block that is being moved also
nonexecutable move(B) & move(B1) if destination(B)=B1.

% two blocks can't be on the same block at the same time
constraint loc(B)=loc(B1) ->> loc(B)=table where B @< B1.
```

Figure 6.4: Blocks World in the language of the new CCALC

```
% File: 'bw-test'

:- include 'bw'.

:- objects
  a,b,c,d                    :: block.

:- query

% initial condition       goal
%
%    a   c                 b   d
%    b   d                 a   c
%  ---------             ---------

 maxstep :: 1..100;

 0: loc(a)=b, loc(b)=table, loc(c)=d, loc(d)=table;
 maxstep: loc(a)=table, loc(b)=a, loc(c)=table, loc(d)=c.
```

Figure 6.5: A query in the Blocks World with four blocks

```
% File: 'bw-neighbor'

:- include 'bw'.

:- constants
  neighbor(block,block) :: sdFluent.

:- variables
  B, B1                 :: block;
  L                     :: location.

% definition of neighbor

caused neighbor(B,B1) if loc(B)=B1 ++ loc(B1)=B.
default -neighbor(B,B1).
```

Figure 6.6: Definition of neighbor

of steps. For instance, every elaboration of MCP formalized in [Lifschitz, 2000] has a solution, so that by specifying the number of steps to try, CCALC found one. However, some other elaborations in the McCarthy's list are not solvable. For example, one elaboration asks whether it is possible to have a solution if there are four missionaries and four cannibals instead of three in each group.

A well-known general method of using invariants helps us prove the unsolvability of planning problems. As discussed in [McCarthy, 1998], we need to check the following three conditions given a property I of states:

- the initial state satisfies I,
- every state that satisfies I is not a goal state,
- in every transition $\langle s, e, s' \rangle$ where s satisfies I, s' also satisfies I.

In terms of transition systems, the conditions ensure that every state that is

reachable from the initial state satisfies the invariant but the goal state does not, so that it is not possible to reach a goal state from the initial state. For instance, an invariant for the unsolvable problem mentioned above is that either the boat is on the first bank on which there are more than 2 missionaries, or the boat is on the second bank on which there are less than 3 missionaries. Once a property I is selected, checking that it satisfies the three conditions above can be reduced to the satisfiability problem. CCALC provides a convenient syntax for doing this as shown in Figure 6.7. The line `maxstep :: any` instructs CCALC that this query is about unsolvability. The next two lines describe the initial and the goal states. An invariant is specified with `invariant:`.

CCALC calls a SAT solver three times to check each of these conditions (*Init* is the formula specified with `0:` in the query and *Goal* is the formula specified with `maxstep:`):

(i) if $Comp(D_0) \cup 0{:}\mathit{Init} \cup 0{:}I$ is satisfiable;

(ii) if $Comp(D_0) \cup 0{:}I \cup 0{:}G$ is unsatisfiable;

(iii) if $Comp(D_1) \cup 0{:}I \cup \neg(1{:}I)$ is unsatisfiable.

In checking each of conditions (ii) and (iii), if the theory is satisfiable, then a SAT solver returns a model, which is a counterexample to the claim.

6.7 Proofs

The proof of Proposition 4 is given in Section 11.2.

Proposition 1 *The models of a definite causal theory are precisely the models of*

```
% File: 'jmc3-test'

:- query
 maxstep :: any;
 0: num(mi,bank1)=4, num(ca,bank1)=4;
 maxstep: num(mi,bank2)=4 & num(ca,bank2)=4;
 invariant:
    num(mi,bank1)+num(mi,bank2)=4 & num(ca,bank1)+num(ca,bank2)=4
    & (loc(boat)=bank1 & num(mi,bank1)>2 ++
       loc(boat)=bank2 & num(mi,bank2)<3).
```

Figure 6.7: Four missionaries and four cannibals—Unsolvable problem

its completion.

Proof Let T be a definite causal theory. Assume that I is a model of T. It follows that, for every rule of the form $\bot \Leftarrow F$ in T, I does not satisfy F, and thus satisfies every formula in the completion of T that is obtained from a constraint. It remains to show that I satisfies the completion formula for every nontrivial atom A. Consider two cases.

Case 1: $A \in T^I$. Since T is definite and $I \models T^I$, T^I is a set of atoms true in I. So I satisfies A, which is the left-hand side of the completion formula for A. Since $A \in T^I$, there is a rule with head A whose body is true in I. Hence I also satisfies the right-hand side of the completion formula for A.

Case 2: $A \notin T^I$. So there is no rule in T with head A whose body is true in I, which shows that I does not satisfy the right-hand side of the completion formula for A. It remains to show that $I \not\models A$. Since T^I is a set of atoms whose unique model is I, every nontrivial atom true in I belongs to T^I. Since A is a nontrivial atom that does not belong to T^I, we can conclude that A is false in I.

Proof in the other direction is similar. ∎

Proposition 2 *For any transition $\langle s, e, s' \rangle$, s and s' are states.*

Proof Let $X = 0{:}s \cup 0{:}e \cup 1{:}s'$ be a model of D_1. We need to show that $0{:}s$ and $0{:}s'$ are models of D_0. By $i{:}\sigma^{fl}$ we denote the set of all constants of the form $i{:}c$ where $c \in \sigma^{fl}$.

To show that $0 : s$ is a model of D_0, observe that D_0 is the part of D_1 consisting of rules (6.10) for static laws with $i = 0$ and rules (6.12). The reduct D_0^X is a set of formulas over $0 : \sigma^{fl}$ and every formula from D_1^X with a constant from $0{:}\sigma^{fl}$ belongs to D_0^X. Since X is the unique model of D_1^X, we can conclude that $0{:}s$ is the unique model of D_0^X. But $D_0^X = D_0^{0:s}$, so that $0{:}s$ is a model of D_0.

Next we show that $0{:}s'$ is a model of D_0. Let T be the part of D_1 consisting of rules (6.10) for static laws with $i = 1$, rules (6.10) for action dynamic laws with $i = 0$, and rules (6.11) with $i = 0$. Let $\Gamma = T^X$. It is straightforward to verify that Γ is a set of formulas over $1{:}\sigma^{fl}$ and that every formula from D_1^X with a constant from $1{:}\sigma^{fl}$ belongs to Γ. Since X is the unique model of D_1^X, we can conclude that $1{:}s'$ is the unique model of Γ. Let Γ_0 be the set of formulas over $0{:}\sigma^{fl}$ obtained from Γ by replacing each time stamp 1: with 0:. Then $0{:}s'$ is the unique model of Γ_0. We need to show that $0{:}s'$ is also the unique model of $D_0^{0:s'}$. Observe first that every formula in $D_0^{0:s'}$ that does not belong to Γ_0 is an atom from $0 : s'$ that came to the reduct from one of the rules (6.12) of D_0. Hence $0 : s'$ satisfies $D_0^{0:s'}$. Due to the presence of rules (6.12) in D_0, any interpretation that satisfies $D_0^{0:s'}$ must agree with $0{:}s'$ on simple fluent constants. On the other hand, the formulas in Γ_0 that do not belong to $D_0^{0:s'}$ do not contain statically determined constants, because their counterparts

in Γ came from the heads of dynamic laws. Consequently any interpretation that satisfies $D_0^{0:s'}$ must agree with $0:s'$ on statically determined fluent constants. It follows that $0:s'$ is the unique model of $D_0^{0:s'}$, so that $0:s'$ is a model of D_0. ∎

Proposition 3 *For any $m > 0$, an interpretation (6.13) of the signature of D_m is a model of D_m iff*

$$\langle s_0, e_0, s_1, e_1, \ldots, s_{m-1}, e_{m-1}, s_m \rangle$$

is a history of D.

Proof We understand the notation $i:\sigma^{fl}$ as in the previous proof, and the meaning of $i:\sigma^{act}$ is similar.

Take a model X of D_m, represent it in the form (6.13), and take any $j \in \{0, \ldots, m-1\}$. We need to show that $0:s_j \cup 0:e_j \cup 1:s_{j+1}$ is a model of D_1.

Let T be the subset of D_m consisting of rules (6.10) for static laws with $i = j+1$, rules (6.10) for action dynamic laws with $i = j$, and rules (6.11) with $i = j$. Let $\Gamma = T^X$. It is straightforward to verify that Γ is a set of formulas over $j:\sigma^{act} \cup\ j+1:\sigma^{fl}$, and that every formula from D_m^X with a constant from $j:\sigma^{act} \cup\ j+1:\sigma^{fl}$ belongs to Γ. Since X is the unique model of D_m^X, it follows that $j:e_j \cup\ j+1:s_{j+1}$ is the unique model of Γ. Let Γ_0 be the set of formulas over $0:\sigma^{act} \cup\ 1:\sigma^{fl}$ obtained from Γ by replacing $j:$ with $0:$ and $j+1:$ with $1:$. Then $0:e_j \cup\ 1:s_{j+1}$ is the only interpretation of $0:\sigma^{act} \cup\ 1:\sigma^{fl}$ that satisfies Γ_0.

The proof of the previous proposition is easily adapted to show that s_j is a state, which is to say that $0:s_j$ is a model of D_0. That is, $0:s_j$ is the unique model of $D_0^{0:s_j} = D_0^{0:s_j \cup\ 0:e_j \cup\ 1:s_{j+1}}$.

It remains to observe that $D_0^{0:s_j \cup\ 0:e_j \cup\ 1:s_{j+1}} \cup \Gamma_0 = D_1^{0:s_j \cup\ 0:e_j \cup\ 1:s_{j+1}}$, so that $0:s_j \cup 0:e_j \cup 1:s_{j+1}$ is the unique model of this set of formulas, and, consequently, a model of D_1.

For the other direction, assume that, for each $j \in \{0, \ldots, m-1\}$, the triple $\langle s_j, e_j, s_{j+1} \rangle$ is a transition. We need to show that the corresponding interpretation X of form (4.13) is a model of D_m.

For each j, let T_j be the subset of D_m consisting of rules (6.10) for static laws with $i = j+1$, rules (6.10) for action dynamic laws with $i = j$, and rules (6.11) with $i = j$. Notice that $D_m = D_0 \cup T_0 \cup \cdots \cup T_{m-1}$. Let $\Gamma_j = T_j^X$. Of course $D_m^X = D_0^X \cup \Gamma_0 \cup \cdots \cup \Gamma_{m-1}$.

For any such j, since $\langle s_j, e_j, s_{j+1} \rangle$ is a transition, $0:s_j \cup 0:e_j \cup 1:s_{j+1}$ is the unique model of of

$$D_1^{0:s_j\ \cup\ 0:e_j\ \cup\ 1:s_{j+1}} = D_0^{0:s_j} \cup T_0^{0:s_j\ \cup\ 0:e_j\ \cup\ 1:s_{j+1}}.$$

Reasoning much as before, it follows that $0:e_j \cup 1:s_{j+1}$ is the unique model of $T_0^{0:s_j\ \cup\ 0:e_j\ \cup\ 1:s_{j+1}}$. This is equivalent to saying that $j:e_j \cup j+1:s_{j+1}$ is the unique model of

$$T_j^{j:s_j\ \cup\ j:e_j\ \cup\ j+1:s_{j+1}} = \Gamma_j.$$

From the previous proposition, we can conclude also that $0:s_0$ is the unique model of D_0^X.

Finally, since $D_m^X = D_0^X \cup \Gamma_0 \cup \cdots \cup \Gamma_{m-1}$, we can conclude that X is the unique model of D_m^X, and thus a model of D_m. ∎

Proposition 5

(*a*) *Let D be an action description, L a static causal law, and ab a Boolean statically determined fluent. If ab does not occur in the heads of any causal laws of D, then the transition system described by $D \cup \{L$ **unless** $ab\}$ is exactly the transition system described by $D \cup \{L\} \cup \{$**caused** $\neg ab\}$.*

(*b*) *Let D be an action description, L a dynamic causal law, and ab a Boolean action constant. If ab does not occur in the heads of any causal laws of D, then the transition system described by $D \cup \{L$ **unless** $ab\}$ is exactly the transition system described by $D \cup \{L\} \cup \{$**caused** $\neg ab$ **after** $\top\}$.*

Proof

(*a*) First we will check that the two transition systems have the same set of states, that is, $(D \cup \{L$ **unless** $ab\})_0$ and $(D \cup \{L\} \cup \{$**caused** $\neg ab\})_0$ have the same models. Let X be a model of $(D \cup \{L$ **unless** $ab\})_0$. i.e.,

$$\begin{aligned} & D_0 \\ \cup\ & (\textbf{caused } F \textbf{ if } G \wedge \neg ab)_0 \\ \cup\ & (\textbf{caused } \neg ab \textbf{ if } \neg ab)_0 \end{aligned}$$

where L is **caused** F **if** G.

It holds that $X(0 : ab) = \mathbf{f}$. Otherwise $(D \cup \{L$ **unless** $ab\})_0^X$ does not contain $0 : ab$ and consequently it contradicts that X is the unique model of $(D \cup \{L$ **unless** $ab\})_0^X$.

It is easy to see that

$$\begin{aligned} (D \cup \{L \textbf{ unless } ab\})_0^X &= D_0^X \cup \{L\}_0^X \cup \{\neg 0 : ab\} \\ &= D_0^X \cup \{L\}_0^X \cup \{\textbf{caused } \neg ab\}_0^X \\ &= (D \cup \{L\} \cup \{\textbf{caused } \neg ab\})_0^X. \end{aligned}$$

Proof in the other direction is similar.

The same reasoning applies to show that the transition systems have the same edges, i.e., $(D \cup \{L \textbf{ unless } ab\})_1$ and $(D \cup \{L\} \cup \{\textbf{caused } \neg ab\})_1$ have the same models.

(b) The proof is similar to the proof of (a).

∎

Proposition 6 *If*

(i) every constant in R is statically determined, and

(ii) for every causal law in D that contains a constant from R in the head, all constants occurring in this law belong to R,

then the mapping $I \mapsto I^R$ is a 1–1 correspondence between the models of D_m and the models of D_m^R.

Proof Using the fact that $s'(c) = s(c)$ for every $c \in R$ and for every transition $\langle s, e, s' \rangle$, it is easy to verify that if I is a model of D_m, then I^R is a model of D_m^R. In other words, the mapping $I \mapsto I^R$ is a function from the set of models of D_m into the set of models of D_m^R. It is also easy to see that the function is 1–1. It remains to show that the function is onto.

Take any model J of D_m^R. Define the interpretation I of the signature of D_m as follows:

$$I(i{:}c) = \begin{cases} J(c) & \text{if } c \in R, \\ J(i{:}c) & \text{otherwise.} \end{cases}$$

Then

$$I(0{:}c) = \cdots = I(m{:}c) \tag{6.30}$$

for all $c \in R$, and $I^R = J$. We will check that I is a model of D_m, that is to say, the only model of D_m^I.

For any formula F of the signature of D_m, let F^R be the result of dropping the time stamps in front of all constants from R in F. It is easy to see that I satisfies a formula F iff J satisfies F^R. It follows that

$$(D_m^R)^J = \{F^R : F \in D_m^I\}. \tag{6.31}$$

Since J is a model of $(D_m^R)^J$, it follows that I is a model of D_m^I. It remains to show that D_m^I has no other models.

Take any model I_1 of D_m^I. By (6.31), I_1^R is a model of $(D_m^R)^J$. Since J is the only model of $(D_m^R)^J$, $I_1^R = J$. Let i_0 be a number from $\{0, \ldots, m\}$. Define the interpretation I_2 of the signature of D_m as follows:

$$I_2(i{:}c) = \begin{cases} I_1(i_0{:}c) & \text{if } c \in R, \\ I_1(i{:}c) & \text{otherwise.} \end{cases}$$

We want to show that I_2 is a model of D_m^I as well. Since I_1 is a model of D_m^I, I_2 satisfies all formulas from D_m^I that do not contain constants from R. Consider a formula from D_m^I that contains at least one constant from R. Since all constants in R are statically determined, this formula has the form $i : F$ for some static causal law (6.8) in D and some time stamp i such that I satisfies $i : G$. By condition (ii), all constants occurring in F, G belong to R. For every constant c from R, I assigns to $i{:}c$ and $i_0{:}c$ the same value; consequently, I satisfies $i_0 : G$, so that $i_0 : F$ belongs to D_m^I. It follows that I_1 satisfies $i_0 : F$. Since I_2 does not differ from I_1

on the constants occurring in this formula, it follows that I_2 satisfies $i_0 : F$. For every constant c from R, I_2 assigns to $i{:}c$ and $i_0{:}c$ the same value; consequently, I_2 satisfies $i : F$. We have established that I_2 indeed satisfies D_m^I.

In view of this fact, it follows from (6.31) that I_2^R satisfies $(D_m^R)^J$. We can conclude that $I_2^R = J$. So, for every $c \in R$, $I_2(0 : c) = J(c)$, and consequently $I_1(i_0 : c) = J(c)$. Since i_0 was arbitrary, we have proved that

$$I_1(0{:}c) = \cdots = I_1(m{:}c) \tag{6.32}$$

for every $c \in R$. And since $I^R = J = I_1^R$, it follows from (6.30) and (6.32) that $I_1 = I$. This shows that D_m^I has no models other than I. ∎

Theorem 1 *For any consistent ADL action description D, any interpretations s, s' of σ^{fl}, and any $e \in \sigma^{act}$, s' is the result of executing e in s according to D iff transition $\langle s, e, s' \rangle$ is a transition of the counterpart of D in language $\mathcal{C}+$.*

Proof Recall that (6.22) stands for

caused $c{=}v$ **if** $c{=}v$ **after** $c{=}v$

caused $a{=}\mathbf{t}$ **if** $a{=}\mathbf{t}$

caused $a{=}\mathbf{f}$ **if** $a{=}\mathbf{f}$

caused $\bot$ **if** $\top$ **after** $a{=}\mathbf{t} \wedge \neg \mathit{Precond}^a$

caused $c{=}v$ **if** $\top$ **after** $a{=}\mathbf{t} \wedge \mathit{Update}_c^a(v)$.

(*Left-to-right*) Assume that s' is the result of executing e in s according to D. For each fluent constant c, the formula $0{:}c{=}s(c)$ is in $D_1^{0:s \cup 0:e \cup 1:s'}$, and the formula $1 : c = s'(c)$ is in $D_1^{0:s \cup 0:e \cup 1:s'}$ (consider two cases, depending on whether $s \models \mathit{Update}_c^e(v)$ for some v); for each action constant a, the formula $0{:}a{=}e(a)$ is in

$D_1^{0:s\cup 0:e\cup 1:s'}$; no other formulas are in the reduct. Consequently, $0:s\cup 0:e\cup 1:s'$ is the only interpretation satisfying the reduct, i.e., $\langle s,e,s'\rangle$ is a transition of D.

(*Right-to-left*) Assume that $\langle s,e,s'\rangle$ is a transition of D. Then $s\models \mathit{Precond}^e$, because otherwise $\bot$ would be in $D_1^{0:s\cup 0:e\cup 1:s'}$. Take a fluent constant c. If, for some v, $s\models \mathit{Update}_c^e(v)$, then $1{:}c{=}v$ is in $D_1^{0:s\cup 0:e\cup 1:s'}$, so that $s'(c)=v$. If, on the other hand, $s\models \neg\exists x\,\mathit{Update}_c^e(x)$ then $s'(c)=s(c)$. Indeed, suppose that $s'(c)\neq s(c)$. Then no formula of the form $1:c=v$ is in $D_1^{0:s\cup 0:e\cup 1:s'}$. But $|\mathit{Dom}|\geq 2$, because $s(c),s'(c)\in \mathit{Dom}$. Consequently, $1:s'$ cannot be the only interpretation satisfying all formulas of the form $1{:}c{=}v$ in $D_1^{0:s\cup 0:e\cup 1:s'}$. ∎

For each interpretation I of σ there is a corresponding interpretation I_c of σ_c such that f or all atoms A common to both signatures

$$I\models A \quad \text{iff} \quad I_c\models A\,,$$

and for all $v\in \mathit{Dom}(c)$

$$I\models c{=}v \quad \text{iff} \quad I_c\models c(v){=}\mathbf{t}\,.$$

The following lemma is easily proved by structural induction.

Lemma 1 *For any formula F and any interpretation I, $I\models F$ iff $I_c\models F_c$.*

Proposition 7 *For any set Γ of formulas and any constant c, the elimination of c from Γ correctly reduces c in Γ.*

Proof Follows from Lemma 1 and the fact that the models of elim_c are precisely the interpretations of σ_c that correspond to an interpretation of σ. ∎

Proposition 8 *For any causal theory T and any constant c, the general elimination of c from T correctly reduces c in T.*

Proof Let T_c be the general elimination of c from T. Because of (6.24), any model of T_c^I, for any interpretation I of σ_c, corresponds to an interpretation of σ. Consider any interpretations I, J of σ, and the corresponding interpretations I_c, J_c of σ_c. It is easy to verify (using Lemma 1) that

$$J \models T^I \quad \text{iff} \quad J_c \models T_c^{I_c}.$$

The result follows easily from these observations. ∎

Proposition 9 *For any causal theory T and any constant c such that (i) $Dom(c)$ has at least two elements, and (ii) every head in which c occurs is an atom, the definite elimination of c from T correctly reduces c in T.*

We begin the proof of Proposition 9 with a lemma.

Lemma 2 *Let T_c be the definite elimination of a constant c from a causal theory T. For any interpretations I, J of σ, and the corresponding interpretations I_c, J_c of σ_c, if $I(c) = J(c)$, then $J \models T^I$ iff $J_c \models T_c^{I_c}$.*

Proof From Lemma 1, $T_c^{I_c} = \{ F_c : F \in T^I \} \cup \{ c(v) = \mathbf{f} : v \neq I(c) \}$.

(*Left-to-right*) Assume that $J \models T^I$. By Lemma 1, $J_c \models \{ F_c : F \in T^I \}$. Since $I(c) = J(c)$, $J_c \models c(v) = \mathbf{f}$ for every v different from $I(c)$. Therefore $J_c \models T_c^{I_c}$.

(*Right-to-left*) Assume that $J_c \models T_c^{I_c}$. Then $J_c \models \{ F_c : F \in T^I \}$, and by Lemma 1, $J \models T^I$. ∎

Proof of Proposition 9 Let T_c be the definite elimination of c from T. We must show that an interpretation is a model of T_c iff it corresponds to a model of T.

(*Left-to-right*) For any interpretation I of σ_c, if T_c^I is satisfiable, then by (6.26) at least one atom $c(v)=\mathbf{t}$ belongs to I, and by (6.25) at most one such atom belongs to I. It follows that any model of T_c corresponds to an interpretation of σ. Assume that I_c is a model of T_c. By Lemma 2, $I \models T^I$. It remains to show that it is the only one. Since I_c is the unique model of $T_c^{I_c}$, it follows that $T_c^{I_c}$ entails $c(v)=\mathbf{t}$ for $v \in Dom(c)$ such that $I(c) = v$. Since any consequent of T in which c occurs is an atom, so is any consequent of T_c in which $c(v)$ occurs. We can conclude that $c(v)=\mathbf{t}$ belongs to $T_c^{I_c}$. By Lemma 1, it follows that $c=v$ is in T^I. Let J be any model of T^I. Then $J \models c=v$, and consequently $I(c) = J(c)$. By Lemma 2, it follows that $J_c \models T_c^{I_c}$. Since I_c is the unique model of $T_c^{I_c}$, $J_c=I_c$. Consequently $I=J$. We proved that I is the only model of T^I.

(*Right-to-left*) Assume that I is a model of T. By Lemma 2, $I_c \models T_c^{I_c}$. Since any formula of T^I in which c occurs must be an atom, and $Dom(c)$ has at least two elements, we can conclude that $c=v$ is in T^I, where v is the element of $Dom(c)$ such that $I \models c=v$. It follows by Lemma 1 that $c(v)=\mathbf{t}$ is in $T_c^{I_c}$. Moreover, by (6.25), $c(v')=\mathbf{f}$ is in $T_c^{I_c}$ for all $v' \in Dom(c)$ such that $v' \neq v$. So any model of $T_c^{I_c}$ corresponds to an interpretation of σ. Let J be any interpretation such that $J_c \models T_c^{I_c}$. Since $T_c^{I_c}$ contains exactly one formula of the form $c(v)=\mathbf{t}$ and contains $c(v')=\mathbf{f}$ for all $v' \in Dom(c)$ such that $v' \neq v$, it follows that $I_c(c(v))=J_c(c(v))$ for all $v \in Dom(c)$, or $I(c)=J(c)$. By Lemma 2, it follows that $J \models T^I$. Since I is a model of T, it follows that $I=J$. Consequently $I_c=J_c$. We proved that I_c is the only model of $T_c^{I_c}$. ∎

Propositions 10 and 11 follow from Propositions 8 and 9 each.

Chapter 7

Representing the Zoo orld in the Language of the Causal Calculator

7.1 Introduction

Research on formalizing commonsense knowledge has been mostly focused on "toy problems," which can be formalized by, say, several lines of axioms, as in the Monkey and Bananas problem. On the other hand, commonsense knowledge that humans have includes many thousands or maybe millions of facts. In this sense we are not close to the goal yet.

The work described in this chapter is intermediate between these two. We formalize a domain which requires several pages of axioms to formalize. Thus testing the the formalization by hand is not feasible, so that an automated system such as

CCALC is essential to test the formalization.

The test domain discussed in this chapter, the Zoo World, is the one of the challenge problems proposed by Erik Sandewall in his Logic Modelling Workshop (LMW)[1]—an environment for communicating axiomatizations of action domains of nontrivial size. The Zoo World consists of several cages and the exterior, gates between them, and animals of several species, including humans. Actions in this domain include moving within and between cages, opening and closing gates, and mounting and riding animals. More details can be found in the next section, which contains extensive quotes from the LMW descriptions of the domain.

In accordance with our goal, we have attempted to translate these descriptions into the input language of CCALC as closely as we could, including the elements that look somewhat arbitrary. One such element in the LMW description of the Zoo World has to do with the "occupancy restriction"—there can be at most one large animal in each position. On the one hand, LMW specifies that this restriction holds even dynamically: a concurrent move, where one animal moves into a position at the same time as another animal moves out of it, is only possible if at least one of the animals is small. On the other hand, the specification tells us that an attempt to mount an animal fails if the animal moves at the same time, in which case "the result of the action is that the human moves to the position where the animal was." Thus a failed mount is an exception to the occupancy restriction. In view of this fact, the occupancy restriction has to be formalized as a defeasible dynamic law. It is interesting to note that expressing such laws in $\mathcal{C}+$ calls for the use of non-exogenous action constants—a new feature of this language, not available in its

[1] `http://www.ida.liu.se/ext/etai/lmw/` .

ancestor $\mathcal{C}$.

We present our formalization of the Zoo World along with detailed comments in Section 7.4, and show how we used CCALC to test it in Section 7.5.

7.2 The Description of the Zoo World

The following is the exact LMW description of the Zoo World that we want to formalize:

> The ZOO is a scenario world containing the main ingredients of a classical zoo: cages, animals in the cages, gates between two cages as well as gates between a cage and the exterior. In the ZOO world there are animals of several species, including humans. Actions in the world may include movement within and between cages, opening and closing gates, feeding the animals, one animal killing and eating another, riding animals, etc.
>
> ... A finite surface area consists of a large number of *positions*. For example, one may let each position be a square, so that the entire area is like a checkerboard. However, the exact shape of the positions is not supposed to be characterized, and the number of neighbors of each position is left open, except that each position must have at least one neighbor. The neighbor relation is symmetric, of course, and the transitive closure of the neighbor relation reaches all positions.
>
> One designated location is called the *outside*; all other locations are called *cages*... The distinction between a 'large' number of positions

and a 'small' number of locations suggests in particular that locations can be individually named under a unique names assumption, that every location is thus named, but on the other hand that at most a few of the positions are named, and that the number of positions is left unspecified in every scenario.

Each position is included in exactly one location. Informally, each cage as well as the outside consists of a set of positions, viewed for example as tiles on the floor. Two locations are neighbors if there is one position in each that are neighbors.

The scenario also contains a small number (in the same sense as above) of gates. Informally, these are to be thought of as gates that can be opened and closed, and that allow passage between a cage and the outside, or between two cages. Formally, each gate is associated with exactly two positions that are said to be at its *sides*, and these positions must belong to different locations.

... Some designated animals will need to be named, but the set of animals in a scenario may be large, and it may not be possible to know them all or to name them all. Animals may be born and may die off over time.

Each animal belongs to exactly one of a number of species. All the species are named and explicitly known. The membership of an animal in a species does not change over time. The species *human* is always defined, and there is at least one human-species animal in each scenario.

Each animal also has the boolean properties *large* and *adult*. Some

species are large, some are not. Adult members of large species are large animals; all other animals are small (non-large).

Each animal has a position at each point in time. Two large animals can not occupy the same position, except if one of them rides on the other (see below).

... Animals can move. In one unit of time, an animal can move to one of the positions adjacent to its present one, or stay in the position where it is. Moves to non-adjacent positions are never possible. Movement is only possible to positions within the same location (for example, within the same cage), and between those two positions that are to the side of the same gate, but only provided the gate is open. Several animals can move at the same time.

Movement actions must also not violate the occupancy restriction: at most one large animal in each position. This restriction also holds within the duration of moves, in the sense that a concurrent move where animal A moves into a position at the same time as animal B moves out of it, is only possible if at least one of A and B is a small animal.

This means in particular that two large animals can not pass through a gate at the same time (neither in the same direction nor opposite directions).

... The following actions can be performed by animals...

- Move To Position. Can be performed by any animal, under the restrictions described above, plus the restriction that a human riding

an animal can not perform the Move-To-Position action (compare below).

- Open Gate. Can be performed by a human when it is located in a position to the side of the gate, and has the effect that the gate is then open until the next time a Close Gate action is performed.
- Close Gate. Can be performed by a human when it is located in a position to the side of the gate, and has the effect that the gate is closed until the next time an Open Gate action is performed.
- Mount Animal. Can be performed by a human mounting a large animal, when the human is in a position adjacent to the position of the animal. The action fails if the animal moves at the same time, and in this case the result of the action is that the human moves to the position where the animal was. If successful, the action results in a state where the human rides the animal. This condition holds until the human performs a Getoff action or the animal performs a Throwoff action.

  When a human rides an animal, the human can not perform the Move action, and his position is the same as the animal's position while the animal moves.
- Getoff Animal to Position. Can be performed by a human riding an animal, to a position adjacent to the animal's present position provided that the animal does not move at the same time. Fails if the animal moves, and in this case the rider stays on the animal.
- Throwoff. Can be performed by an animal ridden by a human, and

results in the human no longer riding the animal and ending in a position adjacent to the animal's present position. The action is nondeterministic since the rider may end up in any such position. If the resultant position is occupied by another large animal then the human will result in riding that animal instead.

7.3 More on the Language of the Causal Calculator

Below we explain the features of the language of CCALC which have not been explained in Section 6.5 but are used in the formalization of the Zoo World.

1. Rigid constants (Section 6.2.6) can be declared as in the following example.

```
:- constants
  sp(animal)            :: species.
```

Function constant `sp` represents an operation that turns an animal into its species.

2. The arguments of constants are supposed to be objects; when a constant appears as an argument of another constant, the former is understood as the value of the constant. For instance, the schema

```
  nonexecutable move(ANML,pos(ANML))
```

("an animal cannot move into its current position") has the same meaning as

```
  nonexecutable move(ANML,P) if pos(ANML)=P
```

where `P` is a `position` variable.

3. Macros can be declared as in the following example.

```
% two locations are neighbors if there is one position in each
%  that are neighbors
:- macros
  neighbor1(#1,#2) ->
       ((#1)\=(#2) & [\/P \/P1 | loc(P)=(#1) & loc(P1)=(#2)
                                             & neighbor(P,P1)]).
```

(`#1`,`#2`,... are parameters for macros.) Upon reading an input file, CCALC replaces every occurrence of a pattern in the left-hand side of `->` with the corresponding instance of the right-hand side.

7.4 Formalization of the Zoo World

Our formalization of the Zoo World shown below is also available online[2].

We distinguish between the general assumptions about the Zoo World quoted in Section 7.2 above, and specific details, such as the "topography" of the zoo (including the number of cages and gates), names of species other than `human`, and so forth. We formalize here the general assumptions only, and leave these details unspecified. A description of all the specifics has to be added to our formalization to get an input file accepted by CCALC. The specific topography used in our computational experiments is described in Section 7.5.

The annotation `(lmw)` found in many comments below refers to the Logic Modelling Workshop description of the Zoo World quoted in Section 7.2.

```
%%% ZOO LANDSCAPE %%%
```

[2] `http://www.cs.utexas.edu/users/tag/ccalc/zoo/` .

```
:- sorts
  position;
  location >> cage;
  gate.

:- variables
  P,P1                              :: position;
  L                                 :: location;
  C                                 :: cage;
  G,G1                              :: gate.

:- constants
% Each position is included in exactly one location (lmw)
  loc(position)                     :: location;
  neighbor(position,position)       :: boolean;
  side1(gate)                       :: position;
  side2(gate)                       :: position;
  opened(gate)                      :: inertialFluent.


default -neighbor(P,P1).

% Each position must have at least one neighbor (lmw)
```

```
constraint [\/P1 | neighbor(P,P1)].

% The neighbor relation is irreflexive
constraint -neighbor(P,P).

% The neighbor relation is symmetric (lmw)
constraint neighbor(P,P1) ->> neighbor(P1,P).

:- objects
% One designated location is called the outside (lmw)
  outside                          :: location.

% All other locations are cages (lmw)
constraint [\/C | L=C] where L\=outside.

% Two positions are the sides of a gate
:- constants
  sides(position,position,gate)     :: boolean.

caused sides(P,P1,G) if side1(G)=P & side2(G)=P1.
caused sides(P,P1,G) if side1(G)=P1 & side2(G)=P.
default -sides(P,P1,G).

% Each gate is associated with exactly two positions that are said to be
%  at its sides, and these positions must belong to different locations
```

```
%  (lmw)
constraint loc(side1(G))\=loc(side2(G)).
```

(As in 2 of Section 7.3, the argument of `loc` is supposed to be an object, not a constant. Here `side1(G)`, `side2(G)` are understood to be the value of each constant.)

```
% No two gates have the same two sides
constraint sides(P,P1,G) & sides(P,P1,G1) ->> G=G1.

% Two positions are neighbors if they are the sides of a gate
constraint sides(P,P1,G) ->> neighbor(P,P1).

% Two positions in different locations are neighbors only if they are the
%  two sides of a gate
constraint loc(P)\=loc(P1) & neighbor(P,P1) ->> [\/G | sides(P,P1,G)].


%%% ANIMALS %%%

:- sorts
  animal >> human;
  species.

:- variables
  ANML,ANML1                          :: animal;
```

```
  H,H1                              :: human;
  SP                                :: species.

:- objects
% One of the species is human (lmw)
  humanSpecies                      :: species.

:- constants
% Each animal belongs to exactly one of a number of species (lmw)
%  Membership of an animal in a species does not change over time (lmw)
  sp(animal)                        :: species;
% Some species are large, some are not (lmw)
  largeSpecies(species)             :: boolean;
% Each animal has a position at each point in time (lmw)
  pos(animal)                       :: inertialFluent(position);
% Boolean property of animals (lmw)
  adult(animal)                     :: boolean;

  mounted(human,animal)             :: inertialFluent.

default largeSpecies(SP).
default adult(ANML).

% Humans are a species called humanSpecies
```

```
caused sp(H)=humanSpecies.
constraint sp(ANML)=humanSpecies ->> [\/H | ANML=H].

:- macros
% Adult members of large species are large animals (lmw)
  large(#1) -> adult(#1) & largeSpecies(sp(#1)).

% There is at least one human-species animal in each scenario (lmw)
constraint [\/H | true].

% Two large animals can not occupy the same position, except if one of them
%  rides on the other (lmw)
constraint pos(ANML)=pos(ANML1) & large(ANML) & large(ANML1)
           ->> [\/H | (H=ANML & mounted(H,ANML1)) ++
                      (H=ANML1 & mounted(H,ANML))] where ANML@<ANML1.
```

(@< is a fixed total order.)

```
%%% CHANGING POSITION %%%

:- constants
accessible(position,position)       :: sdFluent.

caused accessible(P,P1)
       if neighbor(P,P1) & -[\/G | sides(P,P1,G) & -opened(G)].
default -accessible(P,P1).
```

```
% In one unit of time, an animal can move to one of the positions
%  accessible from its present one, or stay in the position where it is.
%  Moves to non-accessible positions are never possible (lmw)
constraint pos(ANML)\=P1 after pos(ANML)=P & -(P=P1 ++ accessible(P,P1)).
```

(The proposition **constraint** F **after** G is an abbreviation for **caused** $\perp$ **if** $\neg F$ **after** G.)

```
% A concurrent move where animal A moves into a position at the same time
%  as animal B moves out of it, is only possible if at least one of A and
%  B is a small animal. (lmw)
%  Exceptions for (failed) mount actions and certain occurrences of
%  throwOff -- when thrown human ends up where another large animal was
%  (see the first two propositions in '%%% ACTIONS %%%')
constraint -(pos(ANML)=P & pos(ANML1)\=P)
           after pos(ANML)\=P & pos(ANML1)=P & large(ANML) & large(ANML1)
           unless ab(ANML).

% Two large animals cannot pass through a gate at the same time
%  (neither in the same direction nor opposite directions) (lmw)
constraint -(pos(ANML)=P1 & pos(ANML1)=P1)
           after pos(ANML)=P & pos(ANML1)=P & sides(P,P1,G)
                 & large(ANML) & large(ANML1) where ANML@<ANML1.
constraint -(pos(ANML)=P & pos(ANML1)=P1)
           after pos(ANML)=P1 & pos(ANML1)=P & sides(P,P1,G)
                 & large(ANML) & large(ANML1) where ANML@<ANML1.
```

```
% While a gate is closing, an animal cannot pass through it
constraint -opened(G) ->> pos(ANML)\=P1
           after pos(ANML)=P & sides(P,P1,G) & opened(G).


%%% ACTIONS %%%

:- variables
  A,A1                                :: exogenousAction.

:- constants
  move(animal,position),
  open(human,gate),
  close(human,gate),
  mount(human,animal),
  getOff(human,animal,position),
  throwOff(animal,human)              :: exogenousAction.

:- macros
% Action #1 is executed by animal #2
  doneBy(#1,#2) ->
    ([\/P | #1==move(#2,P)] ++
     [\/G | #1==open(#2,G) ++ #1==close(#2,G)] ++
     [\/ANML | #1==mount(#2,ANML)] ++
```

```
    [\/ANML \/P | #1==getOff(#2,ANML,P)] ++
    [\/H | #1==throwOff(#2,H)]).
```

(Different from "=" used in an atom, "==" is a comparison operator.)

```
% A failed mount is not subject to the usual, rather strict,
%  movement restriction on large animals
mount(H,ANML) causes ab(H).

% If the position a large human is thrown into was previously occupied by
%  another large animal, the usual movement restriction doesn't apply
throwOff(ANML,H) causes ab(H).
```

(The two propositions above describe exceptional circumstances for the movement restriction between large animals.)

```
% Every animal can execute only one action at a time
nonexecutable A & A1 if doneBy(A,ANML1) & doneBy(A1,ANML1) where A@<A1.

% Direct effect of move action
move(ANML,P) causes pos(ANML)=P.

% An animal can't move to the position where it is now
nonexecutable move(ANML,pos(ANML)).

% A human riding an animal cannot perform the move action (lmw)
nonexecutable move(H,P) if mounted(H,ANML).
```

```
% Effect of opening a gate
open(H,G) causes opened(G).

% A human cannot open a gate if he is not located at a position to the
%  side of the gate (lmw)
nonexecutable open(H,G) if -(pos(H)=side1(G) ++ pos(H)=side2(G)).

% A human cannot open a gate if he is mounted on an animal
nonexecutable open(H,G) if mounted(H,ANML).

% A human cannot open a gate if it is already opened
nonexecutable open(H,G) if opened(G).

% Effect of closing a gate
close(H,G) causes -opened(G).

% A human cannot close a gate if he is not located at a position to the
%  side of the gate (lmw)
nonexecutable close(H,G) if -(pos(H)=side1(G) ++ pos(H)=side2(G)).

% A human cannot close a gate if he is mounted on an animal
nonexecutable close(H,G) if mounted(H,ANML).
```

```
% A human cannot close a gate if it is already closed
nonexecutable close(H,G) if -opened(G).

% When a human rides an animal, his position is the same as the animal's
%  position while the animal moves (lmw)
caused pos(H)=P if mounted(H,ANML) & pos(ANML)=P.

% If a human tries to mount an animal that doesn't change position,
%  mounting is successful
caused mounted(H,ANML) if pos(ANML)=P after pos(ANML)=P & mount(H,ANML).

% The action fails if the animal changes position, and in this case the
%  result of the action is that the human ends up in the position where
%  the animal was (lmw)
caused pos(H)=P if pos(ANML)\=P after pos(ANML)=P & mount(H,ANML).

% A human already mounted on some animal cannot attempt to mount
nonexecutable mount(H,ANML) if mounted(H,ANML1).

% A human can only be mounted on a large animal
constraint mounted(H,ANML) ->> large(ANML).

% A human cannot attempt to mount a small animal (lmw)
nonexecutable mount(H,ANML) if -large(ANML).
```

```
% A large human cannot be mounted on a human
constraint mounted(H,H1) ->> -large(H).

% A large human cannot attempt to mount a human
nonexecutable mount(H,H1) if large(H).

% An animal can be mounted by at most one human at a time
constraint -(mounted(H,ANML) & mounted(H1,ANML)) where H@<H1.

% A human cannot attempt to mount an animal already mounted by a human
nonexecutable mount(H,ANML) if mounted(H1,ANML).

% A human cannot be mounted on a human who is mounted
constraint -(mounted(H,H1) & mounted(H1,ANML)).

% A human cannot attempt to mount an animal if the human is already
%  mounted by a human
nonexecutable mount(H,ANML) if mounted(H1,H).

% A human cannot attempt to mount a human who is mounted
nonexecutable mount(H,H1) if mounted(H1,ANML).

% The getOff action is successful provided that the animal does not move
```

```
%  at the same time.  It fails if the animal moves, and in this case the
%  rider stays on the animal (lmw)
caused pos(H)=P if pos(ANML)=P1 after pos(ANML)=P1 & getOff(H,ANML,P).

caused -mounted(H,ANML) if pos(ANML)=P1
                        after pos(ANML)=P1 & getOff(H,ANML,P).

% The action cannot be performed by a human not riding an animal (lmw)
nonexecutable getOff(H,ANML,P) if -mounted(H,ANML).

% A human cannot attempt to getOff to a position that is not accessible
%  from the current position
nonexecutable getOff(H,ANML,P) if -accessible(pos(ANML),P).

% The throwOff action results in the human no longer riding the animal
%  and ending in a position adjacent to the animal's present position.
%  It is nondeterministic since the rider may end up in any position
%  adjacent to the animal's present position (lmw)
throwOff(ANML,H) may cause pos(H)=P.
throwOff(ANML,H) causes -mounted(H,ANML).

% If the resultant position is occupied by another large animal then the
%  human will result in riding that animal instead (lmw)
caused mounted(H,ANML1) if pos(H)=pos(ANML1) & large(ANML1)
```

```
                    after throwOff(ANML,H) where H\=ANML1.

% The action cannot be performed by an animal not ridden by a human (lmw)
nonexecutable throwOff(ANML,H) if -mounted(H,ANML).

% The actions getOff and throwOff cannot be executed concurrently
nonexecutable getOff(H,ANML,P) & throwOff(ANML,H).
```

7.5 Testing

To test our formalization, we gave CCALC queries and checked that its answers matched our expectations. Queries related to action domains and their CCALC representations are discussed in [Giunchiglia *et al.*, 2004, Sections 3.3, 6]. Besides the representation of the Zoo World shown above, the CCALC input included the description of a specific landscape. The zoo we used for testing is small. It includes 2 locations—a cage and the outside—that are separated by a gate and consist of 4 positions each (Figure 7.1). All positions within the cage are each other's neighbors, as well as all outside positions. The input also included information about the specific animals mentioned in each query.

1. The gate is closed, and Homer, an adult human, is in position 6. *His goal is to mount Jumbo, an adult elephant, which is in position* 3 *and is not going to move around. How many steps are required to achieve this goal?*

This question can be represented by the following CCALC query:

```
:- query
```

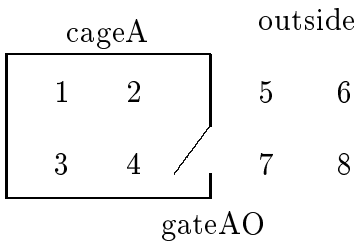

Figure 7.1: A zoo landscape

```
maxstep :: 3..4;
0: -opened(gateAO),
    pos(homer)=6;
maxstep: mounted(homer,jumbo);
T=<maxstep ->> (T: pos(jumbo)=3).
```

(In the last line, `T` is a variable for the initial segment of integers—numbers from `0` to `10`.)

CCALC has determined that the length of the shortest solution is 4. It found a solution in which Homer walks to the gate, opens it, walks into the cage, and then mounts Jumbo.

2. The gate was closed, and Homer was outside; after two steps, he was inside. What can we say about his initial position?

To answer this question, we asked CCALC to find all models satisfying the conditions

```
0: -opened(gateAO),
   loc(pos(homer))=outside;
2: loc(pos(homer))=cageA.
```

CCalc has determined that Homer's only possible initial position is 7. Homer opened the gate and moved to position 4.

3. Initially Homer was outside, and Snoopy, a dog, was inside the cage, with the gate closed. Is it possible that they switched their locations in one step? in two steps? If the elephant Jumbo is substituted for Snoopy, will the answers be the same?

What is essential here is that small animals, unlike elephants, are not affected by the occupancy restriction (Section 7.2); Homer and Snoopy can pass through the gate simultaneously. In response to the query

```
:- query
  maxstep :: 1..2;
  0: -opened(gateA0),
     loc(pos(homer))=outside,
     loc(pos(snoopy))=cageA;
  maxstep:
     loc(pos(homer))=cageA,
     loc(pos(snoopy))=outside.
```

CCalc reported that the length of the shortest solution is 2. In case of Jumbo, CCalc surprised us by discovering that the length of the shortest solution is 4, and not 5 as we had thought. Homer opens the gate, mounts Jumbo (on the other side), dismounts (by either being thrown off or getting off), following which Jumbo moves out of the cage. When we told CCalc that Homer never mounts Jumbo, CCalc agreed that the length of the shortest possible sequence of actions is 5.

4. Can a large animal move into a position at the same time as another large animal moves out of it?

The answer is yes. Although the occupancy restriction applies within the duration of moves (Section 7.2), this scenario is possible in the process of a failed attempt of the first animal to mount the second. There is also the possibility that the first animal is thrown off into the position just vacated by the second.

To investigate this, we asked CCALC whether the following is possible:

```
[\/P | (0: -(pos(homer)=P)) &
       (1: pos(homer)=P) &
       (0: pos(jumbo)=P) &
       (1: -(pos(jumbo)=P))];
0: mounted(homer,silver).
```

(Silver is a horse.) CCALC found a solution in which Silver throws off Homer. Then we replaced the last line of the query with

```
0: [/\ANML| -throwOff(ANML,homer)].
```

CCALC found a solution in which Homer tried to mount Jumbo. On the other hand, a horse cannot possibly move into a position at the same time as an elephant moves out of it. Accordingly, CCALC determined that there is no model satisfying the condition

```
[\/P | (0: -(pos(silver)=P)) &
       (1: pos(silver)=P) &
       (0: pos(jumbo)=P) &
       (1: -(pos(jumbo)=P))].
```

5. In position 1, *Jumbo throws off Homer. What are the possible positions of Jumbo and Homer after that?*

This question illustrates the nondeterministic character of the Throwoff action (Section 7.2). The given assumption can be represented by the condition

```
0: pos(jumbo)=1,
   throwOff(jumbo,homer).
```

According to CCALC, in the models satisfying this condition Homer is thrown into positions 2, 3 and 4; Jumbo always stays in position 1.

Chapter 8

Describing Additive Fluents and Actions in $\mathcal{C}+$

8.1 Concurrent Execution of Actions in $\mathcal{C}+$

Consider a transition system representing the effect of buying a book on the number of books that the person owns (Figure 8.1). It uses two fluent constants—$Has(A)$ (the number of books that Alice has) and $Has(B)$ (the number of books that Bob has)—with the domain $\{0, \ldots, N\}$, where N is a fixed nonnegative integer, and two Boolean action constants—$Buy(A)$ (Alice buys a book) and $Buy(B)$ (Bob buys a book). Every state is represented by two equations showing the values of the fluent constants. Every event is represented by the set of action constants that are mapped to **t**. The loops are labeled by the trivial event $\emptyset$ (no actions are executed). The horizontal edges are labeled by the event in which Alice buys a book and Bob doesn't; along each of the vertical edges, Bob buys a book and Alice doesn't. The

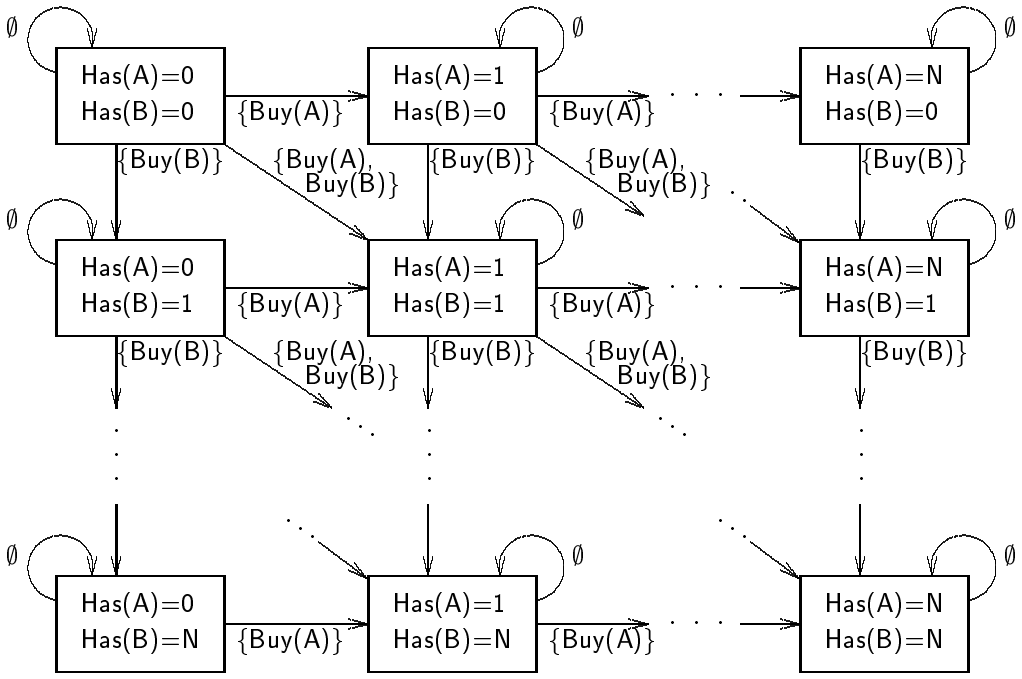

Figure 8.1: A transition system

diagonal edges correspond to Alice and Bob buying books concurrently.

The transition system can be described by the following action description:

$$\begin{array}{l} \textbf{inertial } \mathit{Has}(x) \\ \textbf{exogenous } \mathit{Buy}(x) \\ \mathit{Buy}(x) \textbf{ causes } \mathit{Has}(x)=k+1 \textbf{ if } \mathit{Has}(x)=k \\ \textbf{nonexecutable } \mathit{Buy}(x) \textbf{ if } \mathit{Has}(x)=N \end{array} \tag{8.1}$$

where $x \in \{A, B\}$ and $k \in \{0, ..., N-1\}$.

This action description does not say explicitly that the trivial event $\emptyset$ has no effect on the values of $\mathit{Has}(A)$ and $\mathit{Has}(B)$, or that event $\{\mathit{Buy}(A)\}$ does not affect the value of $\mathit{Has}(B)$. Nevertheless, every edge of the transition system labeled $\emptyset$ is a

loop, and every edge labeled $\{Buy(A)\}$ is horizontal, because of the first line of (8.1) that expresses, under the semantics of $\mathcal{C}+$, the persistence property of $Has(x)$.

Similarly, action description (8.1) does not say anything about the concurrent execution of actions $Buy(A)$ and $Buy(B)$. But the edges labeled $\{Buy(A), Buy(B)\}$ in Figure 8.1 are directed diagonally, in accordance with our commonsense expectations. This fact illustrates the convenience of the approach to concurrency incorporated in the semantics of $\mathcal{C}+$.

However, as discussed in Section 5.6, this built-in mechanism is not directly applicable to the effects of actions on additive fluents, such as the number of books available in the bookstore in the presence of the concurrent execution of buying actions. In this chapter we extend $\mathcal{C}+$ with the additional notation that resolves this difficulty. We introduce here a syntactic construct, **increments**, that allows us to describe the effects of actions on additive fluents. Semantically this construct is treated as "syntactic sugar" on top of $\mathcal{C}+$: the propositions involving that construct are viewed as abbreviations for causal laws of $\mathcal{C}+$. The interpretation of **increments** described below has been used to extend CCALC to cover additive fluents.

8.2 Increment Laws

In our proposed extension of $\mathcal{C}+$, some of the simple fluent constants can be designated as *additive*. The domain of every additive fluent constant is assumed to be a finite set of numbers. We understand "numbers" as (symbols for) elements of any set with an associative and commutative binary operation + that has a neutral element 0.[1] Effects of actions on additive fluents are described in an extended $\mathcal{C}+$ by

[1]The additive group of integers is the main example we are interested in, and this is the case that has been implemented. The max operation on an ordered set with the smallest element is

causal laws of a new kind—"increment laws." Accordingly, we modify the definition of a causal law shown in Section 6.2.1 in two ways. First, in causal laws of the forms (6.8) and (6.9) formula F is not allowed to contain additive fluent constants. Second, we extend the class of causal laws by including *increment laws*—expressions of the form

$$a \textbf{ increments } c \textbf{ by } n \textbf{ if } G \tag{8.2}$$

where

- a is a Boolean action constant,
- c is an additive fluent constant,
- n is a number, and
- G is a formula that contains no Boolean action constants.

We will drop '**if** G' in (8.2) if G is $\top$.

In the next section we define the semantics of the extended $\mathcal{C}+$ by describing a translation that eliminates increment laws in favor of additional action constants.

As an example, consider the effects of actions $Buy(A)$, $Buy(B)$ on the number of books available in the bookstore where Alice and Bob are buying books. A description of these effects in extended $\mathcal{C}+$ is shown in Figure 8.2 (as before, N is a fixed nonnegative integer). The transition system represented by the translation of Figure 8.2 in the non-extended language $\mathcal{C}+$ is depicted in Figure 8.3 (with the auxiliary action constants dropped from the edge labels). The causal laws in Figure 8.2 do not say explicitly that the trivial event $\emptyset$ has no effect on the value of *Available*, or that the concurrent execution of actions $Buy(A)$ and $Buy(B)$ decrements the value

another interesting case.

Notation: x ranges over $\{A, B\}$.

Action constants:	Domains:
$Buy(x)$	Boolean

Additive fluent constant:	Domain:
Available	$\{0, \ldots, N\}$

Causal laws:

$Buy(x)$ **increments** *Available* **by** -1
exogenous $Buy(x)$

Figure 8.2: An action description in extended $\mathcal{C}+$

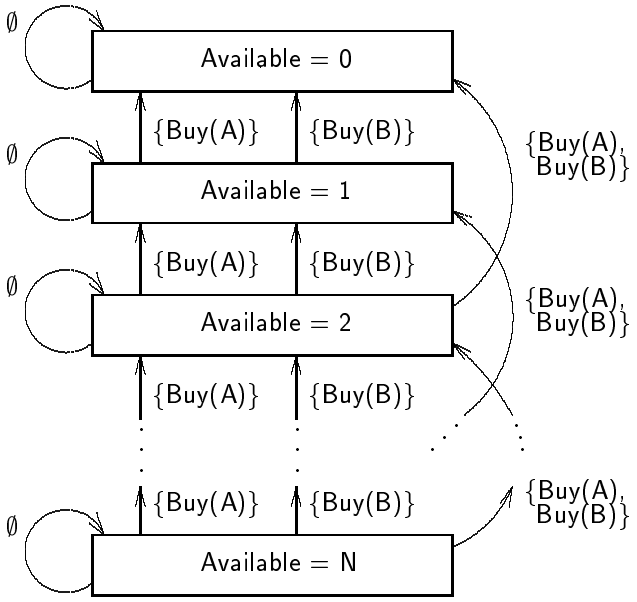

Figure 8.3: The transition system described by Figure 8.2

of this fluent by 2. Nevertheless, every edge of the corresponding transition system labeled $\emptyset$ is a loop, and every edge labeled $\{Buy(A), Buy(B)\}$ goes up 2 levels, in accordance with our commonsense expectations. This happens because Figure 8.2 classifies *Available* as an additive fluent constant.

The causal laws in this action description do not say explicitly that actions $Buy(x)$ are not executable when $Available = 0$, and that actions $Buy(A)$, $Buy(B)$ cannot be executed concurrently when $Available = 1$. This is taken care of by our semantics of increment laws, in view of the fact that the domain of *Available* does not contain negative numbers.

8.3 Translating Increment Laws

Let D be an action description in extended $\mathcal{C}+$. In connection with the increment laws (8.2) in D, the following terminology will be used: about the Boolean action constant a, the additive fluent constant c and the number n in (8.2) we will say that a is a *c-contributing* constant, and that n is a *contribution* of a to c.

The auxiliary action constants introduced in the translation are expressions of the form $Contribution(a, c)$, where c is an additive fluent constant, and a is a c-contributing action constant. The domain of $Contribution(a, c)$ consists of all contributions of a to c and number 0.

To translate the increment laws from D, we

(i) replace each increment law (8.2) in D with the action dynamic law

$$\textbf{caused } Contribution(a, c) = n \textbf{ if } a = \mathbf{t} \wedge G, \tag{8.3}$$

(ii) for every auxiliary constant $\mathit{Contribution}(a, c)$, add the action dynamic law

$$\textbf{caused } \mathit{Contribution}(a, c) = 0 \textbf{ if } \mathit{Contribution}(a, c) = 0, \tag{8.4}$$

(iii) add the fluent dynamic laws

$$\textbf{caused } c = v + \textstyle\sum_a v_a \textbf{ if } \top \textbf{ after } c = v \wedge \bigwedge_a \mathit{Contribution}(a, c) = v_a \tag{8.5}$$

for every additive fluent constant c, every $v \in \mathit{Dom}(c)$, and every function $a \mapsto v_a$ that maps each c-contributing constant a to an element of the domain of $\mathit{Contribution}(a, c)$ so that $v + \sum_a v_a$ is in the domain of c.

The sum and the multiple conjunction in (8.5) range over all c-contributing constants a.

Causal law (8.3) interprets increment law (8.2) as the assertion that executing a (possibly along with other actions) causes constant $\mathit{Contribution}(a, c)$ to get the value n, under some conditions characterized by formula G. Causal laws (8.4) say that the value of this constant is 0 by default, that is to say, when another value is not required by any increment law. Causal laws (8.5) say that the value of an additive fluent constant after an event can be computed as the sum of the value of this constant prior to the event and the contributions of all actions to this constant.

The result of translating increment laws from Figure 8.2 is shown in Figure 8.4. In this case, the translation described above introduces two auxiliary action constants: $\mathit{Contribution}(\mathit{Buy}(A), \mathit{Available})$ and $\mathit{Contribution}(\mathit{Buy}(B), \mathit{Available})$. The domain of each of them has 2 elements: the contribution -1 of $\mathit{Buy}(x)$ to *Available* and number 0.

The edges of the transition system described by Figure 8.4, and the corresponding events, can be computed using the methods presented in Section 6.1.3.

Notation: x ranges over $\{A, B\}$.

Action constants:	Domains:
$Buy(x)$	Boolean
$Contribution(Buy(x), Available)$	$\{-1, 0\}$

Additive fluent constant:	Domain:
$Available$	$\{0, \ldots, N\}$

Causal laws:

caused $Contribution(Buy(x), Available) = -1$ **if** $Buy(x) = \mathbf{t}$
caused $Contribution(Buy(x), Available) = 0$ **if** $Contribution(Buy(x), Available) = 0$
caused $Available = v + v_1 + v_2$ **if** $\top$ **after** $Available = v \wedge$
$Contribution(Buy(A), Available) = v_1 \wedge Contribution(Buy(B), Available) = v_2$
for all $v \in \{0, \ldots, N\}$ and $v_1, v_2 \in \{-1, 0\}$ such that $v + v_1 + v_2 \geq 0$
exogenous $Buy(x)$

Figure 8.4: The result of translating increment laws from Figure 8.2

Every event assigns values to each action constant, including the auxiliary constants $Contribution(Buy(x), Available)$. For instance, the labels

$$\emptyset,\ \{Buy(A)\},\ \{Buy(B)\},\ \{Buy(A), Buy(B)\}$$

in Figure 8.3 represent the following events $E_0, \ldots, E_3$ respectively:

$$\begin{aligned}
&E_0(Buy(A)) = \mathbf{f},\ E_0(Contribution(Buy(A), Available)) = 0,\\
&E_0(Buy(B)) = \mathbf{f},\ E_0(Contribution(Buy(B), Available)) = 0,\\
\\
&E_1(Buy(A)) = \mathbf{t},\ E_1(Contribution(Buy(A), Available)) = -1,\\
&E_1(Buy(B)) = \mathbf{f},\ E_1(Contribution(Buy(B), Available)) = 0,\\
\\
&E_2(Buy(A)) = \mathbf{f},\ E_2(Contribution(Buy(A), Available)) = 0,\\
&E_2(Buy(B)) = \mathbf{t},\ E_2(Contribution(Buy(B), Available)) = -1,\\
\\
&E_3(Buy(A)) = \mathbf{t},\ E_3(Contribution(Buy(A), Available)) = -1,\\
&E_3(Buy(B)) = \mathbf{t},\ E_3(Contribution(Buy(B), Available)) = -1.
\end{aligned} \tag{8.6}$$

For specific values of N, the set of edges can also be generated mechanically, by running CCALC. The translation of Figure 8.2 into the input language of CCALC is shown in Figure 8.5. When instructed to find the edges of the corresponding transition system for `n` equal to 2, CCALC displays 8 solutions, in the following format:

```
Solution 4:

0:  available=2
```

```
% File: 'available'

:- sorts
  person.

:- objects
  a, b              :: person.

:- variables
  X                 :: person.

:- constants
  available         :: additiveFluent(0..n);
  buy(person)       :: exogenousAction.

buy(X) increments available by -1.
```

Figure 8.5: The description from Figure 8.2 in the language of CCALC

```
ACTIONS:  buy(a)  buy(b)

1:  available=0
```

8.4 Reasoning about Money

As an application of these ideas to automated commonsense reasoning, consider the following example:

> I have $6 in my pocket. A newspaper costs $1, and a magazine costs $3. Can I buy two newspapers and one magazine? Or one newspaper and two magazines?

```
% File: 'buying'

:- sorts
  agent;
  resource >> item.

:- variables
  Ag                          :: agent;
  Res                         :: resource;
  It                          :: item;
  M,N                         :: 0..maxAFValue.

:- objects
  buyer,seller                :: agent;
  money                       :: resource.

:- constants
  price(item)                 :: 0..maxAFValue;
  has(agent,resource)         :: additiveFluent(0..maxAFValue);
  buy(item)                   :: exogenousAction;
  howmany(item)               :: attribute(0..maxAFValue) of buy(item).


buy(It) increments has(buyer,It) by N if howmany(It)=N.

buy(It) decrements has(seller,It) by N if howmany(It)=N.

buy(It) increments has(seller,money) by M*N
        if howmany(It)=N & price(It)=M  where M*N =< maxAFValue.

buy(It) decrements has(buyer,money) by M*N
        if howmany(It)=N & price(It)=M  where M*N =< maxAFValue.
```

Figure 8.6: File buying: Buying and selling

```
% File: 'buying-test'

:- maxAFValue :: 7.

:- include 'buying'.

:- objects
  newspaper,magazine          :: item.

price(newspaper)=1.
price(magazine)=3.

% I have $6 in my pocket. A newspaper costs $1, and a magazine
% costs $3. Do I have enough money to buy 2 newspapers and a magazine?
% A newspaper and 2 magazines?

:- query
label :: 1;
maxstep :: 1;
0: has(buyer,money) = 6,
   buy(newspaper),
   howmany(newspaper) = 2,
   buy(magazine),
   howmany(magazine) = 1.

:- query
label :: 2;
maxstep :: 1;
0:  has(buyer,money) = 6,
   buy(newspaper),
   howmany(newspaper) = 1,
   buy(magazine),
   howmany(magazine) = 2.

:- show has(buyer,money).
```

Figure 8.7: File `buying-test`: Do I have enough cash?

These questions are about the executability of some concurrently executed actions, and the answers are determined by the effects of these actions on an additive fluent—the amount of money that I have.

Figure 8.6 describes the relevant properties of buying and selling in the input language of the new CCALC. There are objects of four sorts in this domain: agents, resources, items (to be purchased) and (nonnegative) integers; items are a subset of resources. The buyer and the seller are agents; money is a resource; $0, \ldots,$ `maxInt` are integers. The price of an item is an integer. The number of units of a resource that an agent has is an integer-valued additive fluent. Buying is an exogenous action. The four causal laws that follow these declarations are self-explanatory; `decrements` is an abbreviation defined in terms of `increments`.

Figure 8.7 expresses the two questions stated at the beginning of this section. The first question is whether the transition system contains an edge that begins in a state in which the buyer has \$6, and whose label includes buying two newspapers and one magazine. CCALC responds to this query by finding such an edge.[2]

```
| ?- query 1.
% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 2156 atoms (including new atoms), 8134 clauses
% Writing input clauses... done.  (0.35 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0.11 seconds (prep 0.09 seconds, search 0.02 seconds)

0:  has(buyer,money)=6
```

[2]This example involves the concurrent execution of two actions, but in general the CCALC implementation of additive fluents does not impose any specific restriction on the number of actions that can be executed concurrently.

```
ACTIONS:  buy(newspaper,howmany=2)  buy(magazine,howmany=1)

1:  has(buyer,money)=1
```

Its reply to a similar question about one newspaper and two magazines is negative:

```
| ?- query 2.
% Shifting atoms and clauses... done. (0.01 seconds)
% After shifting: 2156 atoms (including new atoms), 8134 clauses
% Writing input clauses... done.  (0.34 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0.11 seconds (prep 0.09 seconds, search 0.02 seconds)

No solution with maxstep 1.
```

8.5 Reasoning about Motion

Some additive fluents mentioned in the introduction—for instance, the velocity of a particle—are real-valued, rather than integer-valued. CCalc cannot deal with real numbers yet, and its input language does not allow us to express properties of such fluents.

But let's imagine a movable object that is immune to this complication—the spacecraft Integer. Far away from stars and planets, the Integer is not affected by any external forces. As its proud name suggests, the mass of the spacecraft is an integer. For every integer t, the coordinates and all three components of the Integer's velocity vector at time t are integers; the forces applied to the spacecraft by its jet engines over the interval $(t, t+1)$, for any integer t, are constant vectors

whose components are integers as well. If the crew of the Integer attempts to violate any of these conditions, the jets fail to operate!

The motion of the Integer is described in Figure 8.8. The three fluents of the form `pos(axis)` represent the current position of the Integer. The additive fluents `vel(axis)` are the components of its velocity. According to Newton's Second Law, the acceleration created by firing a jet can be computed by dividing the force by the mass of the spacecraft. The first proposition in Figure 8.8 expresses this fact without mentioning the acceleration explicitly, in terms of the change in the velocity over a unit time interval. Symbol `//` stands for integer division; the second proposition tells us that firing a jet is impossible if this division gives a non-zero remainder.

The third proposition says that the position of the spacecraft at time $t+1$ can be computed by adding its average velocity over the interval $(t, t+1)$ to its position at time t. Because the acceleration over this interval is constant, the average velocity is computed as the arithmetic mean of the velocities at times t and $t+1$. We do not include any assumptions about the case when the division by 2 involved in computing this arithmetic mean produces a fraction. The semantics of the language of CCALC guarantee actually that firing jets to achieve this result would be impossible.

Finally, to make planning for the Integer more interesting, we use the constant `maxForce` to limit the power of the jets.

To test our representation, we instruct CCALC to answer the following question (Figure 8.9):

> The mass of the Integer is 1. The Integer has two jets, and the force that can be applied by each jet along each axis is at most 2. The current position of the Integer is $(-1, 0, 1)$, and its current velocity is $(0, 1, 1)$.

```
% File: 'spacecraft'

:- sorts
  integer;
  axis;
  jet.

:- objects
  -maxAFValue..maxAFValue        :: integer;
  x,y,z                          :: axis;
  jet1,jet2                      :: jet.

:- variables
  Ax                             :: axis;
  J                              :: jet;
  F,V,V1,P                       :: integer.

:- constants
  pos(axis)                      :: simpleFluent(integer);
  vel(axis)                      :: additiveFluent(integer);
  fire(jet)                      :: exogenousAction;
  force(jet,axis)                :: attribute(integer) of fire(jet).


fire(J) increments vel(Ax) by V // mass if force(J,Ax) = V.

nonexecutable fire(J) if force(J,Ax) mod mass \= 0.

caused pos(Ax) = P+((V+V1)//2) if vel(Ax) = V1
   after vel(Ax) = V & pos(Ax) = P where (V+V1) mod 2 = 0,
                                         P+((V+V1)//2) >= -maxAFValue,
                                         P+((V+V1)//2) =< maxAFValue.

nonexecutable fire(J) if abs(force(J,Ax)) > maxForce.
```

Figure 8.8: File `spacecraft`: The spacecraft Integer

```
% File: 'spacecraft-test'

:- maxAFValue :: 7.

:- macros
  mass -> 1;
  maxForce -> 2.

:- include 'spacecraft'.

:- query
maxstep :: 1;
 0: (pos(x) = -1  &  pos(y) =  0  &  pos(z) =  1);
 0: (vel(x) =  0  &  vel(y) =  1  &  vel(z) =  1);
 1: (pos(x) =  0  &  pos(y) =  3  &  pos(z) =  1).

:- show pos(Ax); vel(Ax).
```

Figure 8.9: File `spacecraft-test`: How to get there?

How can it get to $(0, 3, 1)$ within 1 time unit?

Here is one of the nine answers produced by CCALC:

```
Solution 1:

0:  pos(x)=-1  pos(y)=0  pos(z)=1  vel(x)=0  vel(y)=1  vel(z)=1

ACTIONS:  fire(jet1,force(x)=1,force(y)=2,force(z)= -2)
fire(jet2,force(x)=1,force(y)=2,force(z)=0)

1:  pos(x)=0  pos(y)=3  pos(z)=1  vel(x)=2  vel(y)=5  vel(z)=-1
```

8.6 Additive Action Constants

Besides additive fluent constants, we can introduce "additive action constants," as follows. Some of the action constants can be designated as additive. Their domains, just as the domains of additive fluent constants, are assumed to consist of numbers. Additive action constants are not allowed in formula F in action dynamic laws (6.9). An increment law is now defined as an expression of the form (8.2), where

- a is a Boolean action constant,
- c is an additive fluent constant or an additive action constant,
- n is a number,
- G is an action formula that contains neither Boolean action constants nor additive action constants.

In the translation of increment laws (8.2) described in Section 8.3, in the case when c is an additive action constant, clause (iii) is modified as follows:

(iii′) add the action dynamic laws

$$\textbf{caused } c = \sum_a v_a \textbf{ if } \bigwedge_a \mathit{Contribution}(a, c) = v_a$$

for every function $a \mapsto v_a$ that maps each c-contributing constant a to an element of the domain of $\mathit{Contribution}(a, c)$ so that $\sum_a v_a$ is in the domain of c.

These causal laws say that the value of an additive action constant during an event can be computed as the sum of the contributions of all actions to this action constant during that event.

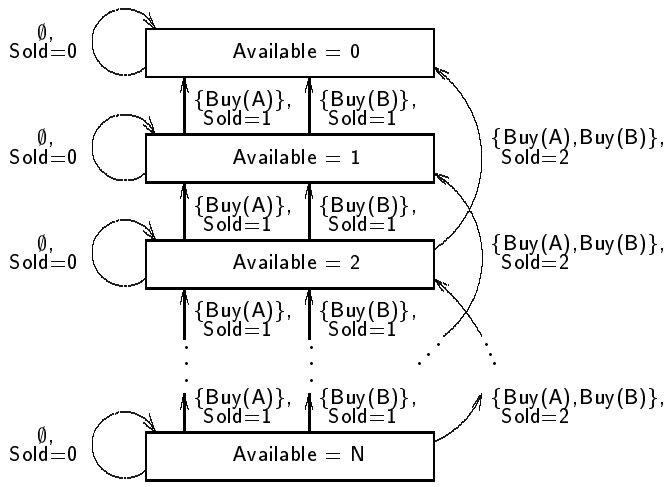

Figure 8.10: A transition system with an additive action constant

Here is an example of the use of additive action constants for representing commonsense knowledge. In Section 8.2 we described how the number of books available in the bookstore is affected by actions of customers. We can ask, on the other hand, how the actions of customers determine the total number of books that are being sold to them at a particular moment; in the case of 2 customers, that number is either 0, 1 or 2. The number of books that are being sold is associated with an event occurring between two successive states, and not with a state. Accordingly, we represent that number by an additive action constant, rather than an additive fluent constant. Extend the action description shown in Figure 8.2 by the additive action constant *Sold* with the domain $\{0, 1, 2\}$, and by the causal law

$$Buy(x) \textbf{ increments } Sold \textbf{ by } 1.$$

The transition system represented by this extension of Figure 8.2 is shown in Figure 8.10.

8.7 Improving Plans

In satisfiability planning, when a plan without concurrent actions is desired, it is usual to make the process of plan generation more efficient by allowing a subset of actions to be executed concurrently as long as that subset is "serializable." In such a plan, the actions that are scheduled for the same time period can be instead executed consecutively, in any order. For example, the restrictions on blocks world plans in file `bw.t` (Figure 4.6) ensure serializability. The 2-step plan shown in Section 6.5 can be turned into a 4-step sequential plan by ordering the actions `move(a,table)` and `move(c,table)` in an arbitrary way, and then ordering `move(b,a)` and `move(d,c)` in an arbitrary way.

Generating serializable solutions is a computationally useful trick, but there is a difficulty associated with it. As observed in [Kautz and Walser, 1999], when the shortest possible serializable plan is found, we cannot generally expect that a sequential plan obtained from it by serialization will be optimal in the sense of the number of steps. Consider, for instance, the blocks world benchmark problem large.c from [Kautz and Selman, 1996] and [Niemelä, 1999]. The problem involves 15 blocks. The shortest serializable solution to this problem consists of 8 steps. Such a solution, found by CCALC using SATO as the search engine, includes 52 moves; there will be only 38 moves, however, if RELSAT is used instead. The shortest serializable solution to large.c found by SMODELS 2.25 on the basis of the formalization given in [Niemelä, 1999] consists of 29 moves. But all these numbers are actually much larger than necessary: as discussed below, there exists a serializable solution to large.c that has length 8 and consists of 18 moves.

Let's define the *cost* of a solution to a blocks world planning problem to be

```
% File: 'bw-cost'

:- include 'bw'.

:- constants
  cost            :: additiveFluent(0..maxAFValue).

move(B) increments cost by 1.
```

Figure 8.11: File `bw-cost`: Computing the cost of a plan

the total number of move actions in it. In the case of a serializable plan, this is the same as the length of a sequential plan obtained from it by serialization. Using the additive fluent mechanism, we can easily characterize the cost of a plan in the language of CCALC (Figure 8.11). Then CCALC can be used to check whether the cost of a plan that it has found is minimal. For instance, in Figure 8.12 we instruct CCALC to find a serializable solution to large.c whose length is 8 and whose cost is at most 18. It produces the following plan:

```
% Calling mChaff...
% Solution time: 9.23 seconds (prep 2.51 seconds, search 6.72 seconds)

0:  cost=0

ACTIONS:  move(c,destination=table)  move(i,destination=table)
move(k,destination=table)

1:  cost=3

ACTIONS:  move(b,destination=table)  move(h,destination=table)
```

```
% File: 'bw-cost-test'

:- maxAFValue :: 19.

:- macros
  length  -> 8;
  maxCost -> 18.

:- include 'bw-cost'.

:- variables
  N                              :: 0..maxAFValue.

:- objects
  a,b,c,d,e,f,g,h,i,j,k,l,m,n,o  :: block.

:- query
maxstep :: (length-1)..length;
0:     cost=0, loc(m)=table, loc(l)=m, loc(a)=l, loc(b)=a, loc(c)=b,
       loc(o)=table, loc(n)=o, loc(d)=n, loc(e)=d, loc(j)=e,
       loc(k)=j, loc(f)=table, loc(g)=f, loc(h)=g, loc(i)=h;
maxstep: cost=<maxCost, loc(e)=j, loc(a)=e, loc(n)=a, loc(i)=d,
       loc(h)=i, loc(m)=h, loc(o)=m, loc(k)=g, loc(c)=k, loc(b)=c,
       loc(l)=b.

:- show cost.
```

Figure 8.12: File: `bw-cost-test`: Finding an economical solution to large.c

```
move(j,destination=table)

2:  cost=6

ACTIONS:  move(e,destination=j)  move(k,destination=g)

3:  cost=8

ACTIONS:  move(a,destination=e)  move(c,destination=k)
move(d,destination=table)

4:  cost=11

ACTIONS:  move(b,destination=c)  move(i,destination=d)

5:  cost=13

ACTIONS:  move(h,destination=i)  move(l,destination=b)
move(n,destination=a)

6:  cost=16

ACTIONS:  move(m,destination=h)

7:  cost=17

ACTIONS:  move(o,destination=m)

8:  cost=18


yes
```

If we make `maxCost` equal to 17 then CCALC tells us that the problem is not

Length of serializable plan	Smallest possible cost
8	18
9	16
10	15
11	15
12	15
13	14
14	14

Figure 8.13: Trade-off between length and cost in solutions to large.c

solvable.

It is interesting to note that large.c has sequential solutions whose length is less than 18. Such solutions cannot be obtained, however, by serializing short concurrent solutions. This kind of trade-off between the length of a serializable solution and the length of the corresponding sequential solution was demonstrated by Kautz and Walser [1999] in the logistics domain. We have used CCALC to investigate this phenomenon in the case of problem large.c. According to the results shown in Figure 8.13, the length of the shortest sequential solution is 14, but such a plan cannot be obtained from a serializable plan whose length is less than 13.

8.8 Properties of Additive Constants

By examining Figure 8.3 in isolation from its symbolic description in Figure 8.2 we can see that the constant *Available* exhibits some features typical for additive fluent constants.

Consider, for instance, the edges that start at the vertex $Available = 2$ and are labeled by the events $\{Buy(A)\}$ and $\{Buy(B)\}$. Each of them leads to

the vertex $Available = 1$, so that each of these two events, when it occurs in the state $Available = 2$, increments the value of *Available* by -1. In accordance with the intuitive idea of an additive fluent, we can expect that the "union" of these events, when it occurs in the same state, will increment the value of *Available* by $(-1)+(-1)$. And this is true, because the edge in Figure 8.3 that starts at the vertex $Available = 2$ and is labeled $\{Buy(A), Buy(B)\}$ leads to the vertex $Available = 0$.

Proposition 12 below generalizes this observation to a class of action descriptions in the language $\mathcal{C}+$ extended as described in Sections 8.2, 8.3. By D we denote any action description in this language.

About events e_0, e_1,...,e_n $(n \geq 0)$ in the transition system represented by D we say that e_0 is a *disjoint union* of e_1,...,e_n if, for every Boolean action constant a,

- if $e_0(a) = \mathbf{t}$ then there exists a unique $i > 0$ such that $e_i(a) = \mathbf{t}$; for this i, $e_0(a') = e_i(a')$ for every non-Boolean action constant a';
- if $e_0(a) = \mathbf{f}$ then, for all $i > 0$, $e_i(a) = \mathbf{f}$.

In the rest of this section we assume that the set of numbers is a commutative group.

Proposition 12 *Let c be an additive fluent constant, let s, $s_0, ..., s_n$ $(n \geq 0)$ be states, and let $e_0, ..., e_n$ be events such that e_0 is a disjoint union of $e_1, \ldots, e_n$. If, for all $i \in \{0, \ldots, n\}$, the transition system represented by D contains an edge that leads from s to s_i and is labeled e_i then*

$$s_0(c) - s(c) = \sum_{i=1}^{n} (s_i(c) - s(c)).$$

The special case corresponding to $n = 0$ tells us that additive fluent constants are not affected by "trivial" events. In this sense, they are similar to the fluent constants for which inertia is postulated.

Corollary 1 *Let e be an event such that for every Boolean action constant a, $e(a) = \mathbf{f}$. If the transition system represented by D contains an edge that leads from a state s to a state s' and is labeled e then, for any additive fluent constant c, $s'(c) = s(c)$.*

The special case corresponding to $n = 1$ implies that the effects of any set of actions on an additive fluent is deterministic:

Corollary 2 *If the transition system represented by D contains an edge that leads from a state s to a state s_0 and is labeled e, and an edge that leads from s to a state s_1 and is also labeled e, then, for any additive fluent constant c, $s_0(c) = s_1(c)$.*

Here are the counterparts of the three facts stated above for additive action constants:

Proposition 13 *Let c be an additive action constant, let s be a state, and let $e_0, ..., e_n$ $(n \geq 0)$ be events such that e_0 is a disjoint union of $e_1, \ldots, e_n$. If, for all $i \in \{0, \ldots, n\}$, the transition system represented by D contains an edge that starts at s and is labeled e_i then*

$$e_0(c) = \sum_{i=1}^{n} e_i(c).$$

Corollary 3 *Let e be an event occurring in the transition system represented by D. If, for every Boolean action constant a, $e(a) = \mathbf{f}$ then, for any additive action constant c, $e(c) = 0$.*

Corollary 4 *Let s be a state, and let e_0, e_1 be events such that for every non-additive action constant a, $e_0(a) = e_1(a)$. If the transition system represented by D contains an edge that starts at s and is labeled e_0, and an edge that starts at s and is labeled e_1, then, for any additive action constant c, $e_0(c) = e_1(c)$.*

8.9 Discussion

In this chapter we showed how an implemented, declarative language for describing actions can be used to talk about the effects of actions on additive fluents. This was accomplished by extending the syntax of the action language $\mathcal{C}+$ from [Giunchiglia *et al.*, 2004] by **increment** laws and by showing how to treat these laws as abbreviations.

It is interesting to note that this treatment of additive fluents would have been impossible if, instead of $\mathcal{C}+$, we used its predecessor $\mathcal{C}$. Non-Boolean, non-exogenous action constants such as $Contribution(a, c)$, and action dynamic laws such as (8.3) and (8.4) are among the features of $\mathcal{C}+$ that were not available in $\mathcal{C}$.

In literature on planning, fluents with numerical values are often referred to as "resources" [Koehler, 1998]. The concurrent execution of the actions that involve resources is usually limited to the "serializable" case, when all ways of sequencing the concurrent actions are well-defined and equivalent. This condition is not satisfied, however, for many uses of additive fluents, including the space travel example (Section 8.5). For instance, in the spacecraft example, firing the jets in one direction and then in the other direction is not the same as firing them concurrently. This example shows that firing jets is not serializable.

8.10 Proofs

Proposition 12 *Let c be an additive fluent constant, let s, $s_0, ..., s_n$ $(n \geq 0)$ be states, and let $e_0, ..., e_n$ be events such that e_0 is a disjoint union of $e_1, \ldots, e_n$. If, for all $i \in \{0, \ldots, n\}$, the transition system represented by D contains an edge that leads from s to s_i and is labeled e_i then*

$$s_0(c) - s(c) = \sum_{i=1}^{n} (s_i(c) - s(c)).$$

Proof Assume that $\langle s, e_0, s_0\rangle$,$\langle s, e_1, s_1\rangle$,...,$\langle s, e_n, s_n\rangle$ are transitions. Consider the reduct $D_1^{0{:}s \cup 0{:}e_i \cup 1{:}s_i}$ where $i > 0$. Since (8.3) and (8.4) are the only causal laws in D that contain constants of the form $\mathit{Contribution}(a, c)$ in the heads, there exists a unique v_a such that $0 : \mathit{Contribution}(a, c) = v_a \in D_1^{0{:}s \cup 0{:}e_i \cup 1{:}s_i}$ for every $0 : \mathit{Contribution}(a, c)$ (Otherwise $\langle s, e_i, s_i\rangle$ would not be a transition). If there is a causal law (8.3) in D such that $e_i(a) = \mathbf{t}$, $e_i \models G$, $s \models H$, then $v_a = n$, a contribution of a to c. Otherwise $v_a = 0$. Since $0 : e_i$ satisfies every atom $0 : \mathit{Contribution}(a, c)$, $(0 : e_i)(0 : \mathit{Contribution}(a, c)) = v_a$. Let $c_i = \sum_a v_a$.

Note that (8.5) is the only causal law in D that contains c in the head. So the reduct contains a unique atom

$$1 : c = (0 : s)(0 : c) + \sum_a (0 : e_i)(0 : \mathit{Contribution}(a, c))$$

for constant $1 : c$. Since $1 : s_i$ satisfies the atom, it follows that $s_i(c) = s(c) + c_i$, or $c_i = s_i(c) - s(c)$.

Now consider the reduct $D_1^{0{:}s \cup 0{:}e_0 \cup 1{:}s_0}$. Again since (8.3) and (8.4) are the only causal laws in D that contain constants of the form $\mathit{Contribution}(a, c)$ in the

heads, there exists a unique v_a such that $0\!:\!\mathit{Contribution}(a,c) = v_a \in D_1^{0:s \cup 0:e_0 \cup 1:s_0}$ for every $0\!:\!\mathit{Contribution}(a,c)$. Notice that

- $v_a = (0\!:\!e_i)(0\!:\!\mathit{Contribution}(a,c))$ for any e_i such that $e_i(a) = \mathbf{t}$ if $e_0(a) = \mathbf{t}$ (indeed there is a unique such e_i), and
- $v_a = 0$ if $e_0(a) = \mathbf{f}$.

(Indeed, suppose $e_0(a) = \mathbf{t}$. There exists a unique e_i such that $e_i(a) = \mathbf{t}$. If there is a causal law (8.3) in D such that $e_0(a) = \mathbf{t}$, $e_0 \models G$, $s \models H$, then $e_i(a) = \mathbf{t}$, $e_i \models G$ also. So $0 : \mathit{Contribution}(a,c) = n$ belongs to both $D_1^{0:s \cup 0:e_0 \cup 1:s_0}$ and $D_1^{0:s \cup 0:e_i \cup 1:s_i}$. If $e_0(a) = \mathbf{t}$, but either $e_0 \not\models G$ or $s \not\models H$, then either $e_i \not\models G$ or $s \not\models H$. So according to (8.4), $0\!:\!\mathit{Contribution}(a,c) = 0$ belongs to both $D_1^{0:s \cup 0:e_0 \cup 1:s_0}$ and $D_1^{0:s \cup 0:e_i \cup 1:s_i}$.)

Since $0\!:\!e_0$ satisfies the atoms, it follows that, for every a,

$$(0\!:\!e_0)(0\!:\!\mathit{Contribution}(a,c)) = (0\!:\!e_i)(0\!:\!\mathit{Contribution}(a,c))$$

for any e_i such that $e_i(a) = \mathbf{t}$ if $e_0(a) = \mathbf{t}$ and

$$(0\!:\!e_0)(0\!:\!\mathit{Contribution}(a,c)) = 0$$

if $e_0(a) = \mathbf{f}$.

Note that (8.5) is the only causal law in D that contains c in the head. So the reduct contains a unique atom

$$1\!:\!c = (0\!:\!s)(0\!:\!c) + \sum_a (0\!:\!e_0)(0\!:\!\mathit{Contribution}(a,c))$$

for constant $1\!:\!c$.

We see that

$$\begin{aligned}\sum_a (0{:}e_0)(0{:}\,\mathit{Contribution}(a,c)) &= \sum_{e_0(a)=\mathbf{t}} (0{:}e_0)(0{:}\,\mathit{Contribution}(a,c)) \\ &= \sum_{1\le i\le n} \sum_{e_i(a)=\mathbf{t}} (0{:}e_i)(0{:}\,\mathit{Contribution}(a,c)) \\ &= \sum_{1\le i\le n} c_i = \sum_{1\le i\le n} (s_i(c) - s(c))\end{aligned}$$

Since $1{:}s_0$ satisfies the atom, it follows that

$$s_0(c) = s(c) + \sum_{1\le i\le n} (s_i(c) - s(c))$$

∎

The proof of Proposition 13 is similar to the proof of Proposition 12.

Chapter 9

Elaborations of the Missionaries and Cannibals Puzzle

As discussed in Section 2.5, McCarthy used elaborations of the Missionaries and Cannibals Puzzle to illustrate the idea of elaboration tolerance. Lifschitz [2000] showed how to formalize McCarthy's elaborations of MCP in an early version of CCalc. His representation did not introduce names for individual missionaries or cannibals; rather, a state was described in terms of the number of members of each group on each bank of the river. As noted in Section 5.6, the "difficult" concurrency identified in that paper has led to the investigation of additive fluents (Chapter 8).

Our formalization presented in this chapter overcomes several limitations of Lifschitz's formalization thanks to the improvements of the language of CCalc. This includes the implementations of additive fluents, defeasible causal laws, and attributes. Rather than presenting all elaborations, we list some of them which illustrate these points.

9.1 Formalization of the Basic Problem

As in [Lifschitz, 2000], we start with formalizing the parts that are common for all elaborations. File `common1` describes the action of crossing using an attribute that denotes the destination.

```
% File 'common1'

:- sorts
  vessel;
  location.    % objects of these sorts should be defined elsewhere

:- variables
  V                      :: vessel;
  L,L1                   :: location.

:- constants
  loc(vessel)            :: inertialFluent(location);
  cross(vessel)          :: exogenousAction;
  to(vessel)             :: attribute(location) of cross(vessel).

cross(V) causes loc(V)=L if to(V)=L unless ab1(V,L).

nonexecutable cross(V) if to(V)=loc(V) unless ab2(V).
```

The line

```
  to(vessel)              :: attribute(location) of cross(vessel)
```

declares that `to` is an attribute of action `cross` whose value is a location. The attribute describes the destination of crossing.

All causal laws in the file are made defeasible. In each elaboration later, if necessary, some of the laws here will be retracted.

File `common2` extends `common1` by introducing new attributes, `howmany(vessel,group)`, that denote how many members of various groups are crossing. As discussed in Section 5.6, the number of members of a group `G` in a location `L` should be treated as an additive fluent to handle "difficult" concurrency. `afValue` is a predefined sort in CCALC that ranges over numbers an additive fluent can take.

```
% File 'common2'

:- include 'common1'.

:- sorts
  group.                % group objects should be defined elsewhere

:- variables
  N,N1                          :: afValue;
  G                             :: group.

:- constants
  num(group,location)           :: additiveFluent(afValue);
  howmany(vessel,group)         :: attribute(afValue) of cross(vessel).
```

```
cross(V) increments num(G,L) by N if to(V)=L & howmany(V,G)=N
   unless ab3(V,G,L).
cross(V) decrements num(G,L) by N if loc(V)=L & howmany(V,G)=N
   unless ab4(V,G,L).
```

File `basic` describes the specifics about the original Missionaries and Cannibals Puzzle, including objects such as the boat, banks, and groups, and constraints such as missionaries should not be outnumbered. This file will be included in all elaborations. Again, all assertions in it are made defeasible.

```
% File 'basic'

:- include 'common2'.

:- objects
  boat                    :: vessel;
  bank1, bank2            :: location;
  mi,ca                   :: group.

:- constants
  capacity(vessel)        :: 1..maxCapacity.

exogenous capacity(V) unless ab5(V).

:- macros
```

```
  outnumbered(#1,#2)                        % #1 missionaries are
      -> (#2 > #1) & (#1 > 0).              % outnumbered by #2 cannibals

% missionaries should not be outnumbered in any location
constraint -outnumbered(num(mi,L),num(ca,L)) unless ab6(L).

% additional preconditions for crossing:
%   someone should be in the boat
nonexecutable cross(V) if howmany(V,mi)+howmany(V,ca)=0 unless ab7(V).

%   but not too many
nonexecutable cross(V)
   if howmany(V,mi)+howmany(V,ca) > capacity(V) unless ab8(V).

% missionaries should not be outnumbered on the way
nonexecutable cross(V) if outnumbered(howmany(V,mi),howmany(V,ca))
   unless ab9(V).

% boat capacity
constraint capacity(boat)=2 unless ab10.
```

To test this formalization, we use the following file `basic-test`. Test files for other elaborations which ask to find plans are similar to this one.

```
% File 'basic-test': original MCP
```

```
:- maxAFValue :: 3.

:- macros
  maxCapacity -> 2.

:- include 'basic'.

:- query
 maxstep :: 10..11;
 0: num(mi,bank1)=3, num(ca,bank1)=3, num(mi,bank2)=0, num(ca,bank2)=0,
    loc(boat)=bank1;
 maxstep: num(mi,bank2)=3, num(ca,bank2)=3.
```

The directive `maxAFValue` specifies the maximum value an additive fluent can take. It also instructs CCALC that additive fluents will be used.

The query instructs CCALC to try to find a plan of length 10 and if there is no such plan, try length 11. Since the shortest step solution for the basic problem involves at least 11 steps, CCALC answered that there is no plan of length 10, and returned a plan of length 11. The solution returned by CCALC, along with solutions for other elaborations, is shown in Appendix A.

9.2 Two Boats

Before presenting formalizations of McCarthy's elaborations, let us consider a simple elaboration in which we allow one more boat which holds only one person. This

modification is more difficult to formalize than the original form of MCP. Besides the difficulty described in Section 5.6, there are other difficulties. Consider the original form of the problem, and imagine that there is a single cannibal with the boat on the left bank. Our postulates should make it impossible, of course, for *two* cannibals to cross in this state. In the absence of a second boat, we don't have to worry about this: two cannibals leaving would have made the number of cannibals on the left bank negative, which is impossible. With two boats, this reasoning does not apply any more, because a cannibal crossing simultaneously in the opposite direction would make the number of cannibals on the left bank equal to 0, which is a legal value. To prohibit such actions, we need to say that the total number of members of a group leaving a location does not exceed the number of members of the group in that location. This is expressed in the formalization below using additive action constant `departing(G,L)`: the total number of members of group `G` who are departing from location `L`.

Another problem which is similar to the above has to do with a constraint on the number of missionaries and cannibals during an event. Imagine that there are two missionaries and two cannibals on the left bank, and only one of the missionaries is leaving. This should be prohibited and indeed if there were only one boat, this would be achieved by the first `constraint` proposition in File `basic`. However, with two boats, if the third missionary arrives into the location simultaneously, the `constraint` proposition does not prohibit the event. Besides the constraint on the number of groups in a location in each state, we need to represent a similar constraint during an event. Below this is expressed using `staying(G,L)`, a macro defined in terms of `departing(G,L)`.

```
% File 'departing'

:- constants
  departing(group,location)  :: additiveAction(afValue).

:- macros
  staying(#1,#2) -> num(#1,#2)-departing(#1,#2).

cross(V) increments departing(G,L) by N if loc(V)=L & howmany(V,G)=N
    unless ab11(V,G,L).

% the number of people departing from a location does not exceed the number
%  of people there
always departing(G,L)=<num(G,L) unless ab13(G,L).

% the missionaries staying in a location should not be outnumbered by
%  the cannibals there.
always staying(mi,L)\=0 ->> staying(mi,L)>=num(ca,L) unless ab12(L).
```

File `two-boats` below includes `departing`, and introduces one more boat, `boat1`, that holds at most one person. Given a query similar to `basic-test` above, CCALC has determined that a solution requires at least 7 steps, and returned a plan of that length.

```
% File 'two-boats'
```

```
:- macros
  maxInt -> 3.

:- include 'basic'; 'departing'.

:- objects
  boat1     :: vessel.

caused capacity(boat1)=1 unless ab14.
```

9.3 Four Missionaries and Four Cannibals

There are four missionaries and four cannibals. The problem is now unsolvable.

We simply use a query of the new form available in CCALC, which was given in Section 6.6

9.4 Boat Can Carry Three

The boat can carry three. Five pairs can cross, but not six. The assumption in File `basic` that the boat can carry only two people should be retracted.

```
% File 'jmc4': McCarthy's Elaboration No. 4

% retract constraint capacity(boat)=2 unless ab10.
```

```
:- include 'basic'.

caused ab10.

constraint capacity(boat)=3 unless ab14.
```

CCALC has determined that at least 11 steps are required if there are five pairs, and returned a plan of that length. It also verified that no solution exists no matter how many steps are given if there are six pairs.

```
% File 'jmc4-test'

:- maxAFValue :: 6.

:- macros
  maxCapacity -> 3.

:- include 'jmc4'.

% Five pairs can cross.
:- query
 label :: 1;
 maxstep :: 10..11;
 0: num(mi,bank1)=5, num(ca,bank1)=5, num(mi,bank2)=0, num(ca,bank2)=0,
    loc(boat)=bank1;
 maxstep:
```

```
    num(mi,bank2)=5, num(ca,bank2)=5.


% Six pairs can't cross.
:- query
 label :: 2;
 maxstep :: any;
 0: num(mi,bank1)=6, num(ca,bank1)=6,  num(mi,bank2)=0, num(ca,bank2)=0,
    loc(boat)=bank1;
 maxstep:
    num(mi,bank2)=6, num(ca,bank2)=6;
 invariant:
    num(mi,bank1)+num(mi,bank2)=6  & num(ca,bank1)+num(ca,bank2)=6
    & (loc(boat)=bank1 & num(mi,bank1)>=4
       ++ loc(boat)=bank2 & num(mi,bank2)=<3).
```

9.5 Converting Cannibals

Three missionaries alone with a cannibal can convert him into a missionary. Lifschitz [2000] noted:

> Do we allow crossing the river and converting a cannibal to occur in parallel? Can a solution begin, for instance, with two cannibals crossing to Bank 2 while the third cannibal is being converted into a missionary on Bank 1? If yes, this is an example of "difficult" concurrency (Section 5) that the approach of this paper does not allow.

Here we do allow the concurrency of this kind. The problem can be solved in 9 steps, one step shorter than the solution reported in [Lifschitz, 2000].

```
% File 'jmc11': McCarthy's elaboration No. 11

:- include 'basic'; 'departing'.

:- constants
  convert(location)     :: exogenousAction.

convert(L) increments num(mi,L) by 1 unless ab14(L).
convert(L) decrements num(ca,L) by 1 unless ab15(L).

% converting is possible only if there are three missionaries and only one
% cannibal in the bank
always convert(L) ->> (staying(mi,L)>=3 & staying(ca,L)=1) unless ab16(L).
```

9.6 Walking on Water

One of the missionaries is Jesus Christ, who can walk on water. This is similar, but not quite equivalent to the elaboration in Section 9.2.

We treat `jc` as a singleton subset of group `mi`. The fact that `jc` is a subset of `mi` is expressed by the postulates that prohibit the states in which the number of `jc` is greater than the number of `mi` in the same location, and that prohibit the events in which the number of `jc` crossing is greater than the number of `mi` crossing.

Since walking and crossing may be executed at the same time, this is another instance of "difficult" concurrency. CCALC has determined that the shortest step solution involves at least 7 steps, and returned a plan of that length.

```
% File 'jmc10': McCarthy's Elaboration No. 10

:- include 'basic'; 'departing'.

:- objects
  jc                          :: group.

:- constants
  walk                        :: exogenousAction;
  walk_to                     :: attribute(location) of walk.

% jc is a subgroup of missionaries
constraint num(jc,L)=<num(mi,L) unless ab14(L).
nonexecutable cross(V) if howmany(V,jc)>howmany(V,mi) unless ab15(V).

% jc can be present at most one location
constraint num(jc,L)=1 ->> num(jc,L1)=0 where L\=L1.
nonexecutable walk if howmany(V,jc)>0 unless ab17(V).

% cannot walk to the same location
nonexecutable walk_to=L if num(jc,L)>0 unless ab18(L).
```

```
walk increments num(mi,L) by 1 if walk_to=L unless ab19(L).
walk decrements num(mi,L) by 1 if walk_to=L1 & L\=L1 unless ab20(L,L1).

walk increments num(jc,L) by 1 if walk_to=L unless ab21(L).
walk decrements num(jc,L) by 1 if walk_to=L1 & L\=L1 unless ab22(L,L1).

walk increments departing(mi,L) by 1 if walk_to=L1 & L\=L1 unless ab23(L,L1).
walk increments departing(jc,L) by 1 if walk_to=L1 & L\=L1 unless ab23(L,L1).
```

9.7 The Bridge

There is a bridge, wide enough for two to cross at once. This is another instance of "difficult" concurrency since using the bridge and the boat concurrently affects the number of people in a location at the same time.

CCALC returned a shortest step solution that involves 4 steps, one step shorter than the solution reported in [Lifschitz, 2000].

```
% File 'jmc13':  McCarthy's elaboration No. 13

:- include 'basic'; 'departing'.

:- constants
  useBridge                          :: exogenousAction;
  useBridge_from,
```

```
  useBridge_to                      :: attribute(location) of useBridge;
  useBridge_howmany(group)          :: attribute(afValue) of useBridge.

useBridge increments num(G,L) by N
   if useBridge_to=L & useBridge_howmany(G)=N unless ab14(G,L).
useBridge decrements num(G,L) by N
   if useBridge_from=L & useBridge_howmany(G)=N unless ab15(G,L).
useBridge increments departing(G,L) by N
   if useBridge_from=L & useBridge_howmany(G)=N unless ab16(G,L).

nonexecutable useBridge if useBridge_from=L & useBridge_to=L1
     unless ab17(G,L) where -((L=bank1 & L1=bank2) ++ (L=bank2 & L1=bank1)).

always useBridge ->> useBridge_howmany(mi)+useBridge_howmany(ca)>0 &
                     useBridge_howmany(mi)+useBridge_howmany(ca)=<2
    unless ab18.
```

Chapter 10

Loop Formulas for Causal Logic

By adding so-called "loop formulas" to completion, Lin and Zhao ensured that the answer sets of a normal logic program are exactly the models of the modified completion. This idea has been extended to more general classes of logic programs, such as programs that allow classical negation, infinite programs, and programs with nested expressions [Lee and Lifschitz, 2003; Lee, 2005]. In this chapter we show how to reduce the general case of causal theories to propositional formulas using the idea of loop formulas.

For simplicity, we limit our attention to Boolean causal theories, that is, to causal theories in the sense of Section 3.3. From Proposition 8 (Section 6.4.2) we know that, in principle, any causal theory can be reduced to a theory of this kind.

10.1 Review of the Lin/Zhao Theorem

Let Π be a normal program (Section 3.1). The *positive dependency graph* of Π is the directed graph such that

Figure 10.1: The dependency graph of Π_1

- its vertices are the atoms occurring in Π, and
- its edges go from p_1 to $p_2, \ldots p_m$ for each rule (3.1) of Π.

A nonempty set L of atoms is called a *loop* of Π if, for every pair p_1, p_2 of atoms in L, there exists a path from p_1 to p_2 in the positive dependency graph of Π such that all vertices in this path belong to L. A loop is called *trivial* if the loop consists of a single atom such that there is no edge from the atom to itself in the positive dependency graph.[1]

For example, consider the following program Π_1:

$$p \leftarrow q$$

$$q \leftarrow p$$

$$r \leftarrow s$$

$$s \leftarrow r$$

$$p \leftarrow not\ r$$

$$r \leftarrow not\ p.$$

The positive dependency graph of Π_1, shown in Figure 10.1, has six loops: $\{p\}$, $\{q\}$, $\{r\}$, $\{s\}$, $\{p, q\}$, $\{r, s\}$.

[1]The example of a singleton loop shows that a loop of Π does not necessarily correspond to a loop (or cycle) of the positive dependency graph of Π in the sense of graph theory. Lin and Zhao [2004] did not allow paths of length 0.

For any set Y of atoms, the *external support formula* for Y is the disjunction of the conjunctions

$$B \wedge F \tag{10.1}$$

for all rules (3.2) of Π such that

- $p_1 \in Y$, and
- $B \cap Y = \emptyset$.

We denote this external support formula by $ES_{\Pi,Y}$. The *(disjunctive) loop formula* of a loop L for Π is the formula

$$\bigvee L \ \supset \ ES_{\Pi,L}.^2 \tag{10.2}$$

By $LF(\Pi)$ we denote the set of formulas (10.2) for all nontrivial loops L of Π.

Theorem [Lin and Zhao, 2004, Theorem 1] *For any normal program Π, a set of atoms is an answer set of Π iff it is a model of $Comp(\Pi) \cup LF(\Pi)$.*

If we include loop formulas for trivial loops, we can reformulate the theorem as follows without referring to completion. In the following we identify Π with a propositional theory by identifying '*not*' with '$\neg$', ',' with '$\wedge$', ';' with '$\vee$', and '$\leftarrow$' with implication.

Corollary [Lee, 2005] *For any normal program Π, a set of atoms is an answer set of Π iff it is a model of*

$$\Pi \ \wedge \bigwedge_{L \text{ is a loop of } \Pi} \Big(\bigvee L \supset ES_{\Pi,L} \Big).$$

[2] When $\bigvee$ is applied to a set L as in the antecedent of this formula, it stands for the disjunction of all elements of L.

10.2 Loop Formulas for Causal Theories in Canonical Form

10.2.1 Main Theorem for Canonical Theories

Recall that a (Boolean) causal rule is an expression of the form

$$F \Leftarrow G,$$

where F, G are propositional formulas.

If the head of a causal rule is a conjunction, then the rule can be broken into simpler rules: replacing a rule

$$F \wedge G \Leftarrow H$$

in a causal theory by the rules

$$F \Leftarrow H,\ G \Leftarrow H$$

does not change the set of models. This fact allows us to replace any rule in a causal theory by several rules whose heads are clauses. We will call such causal rules *canonical*.

A *canonical causal theory* is a finite set of canonical causal rules.

Let T be a canonical causal theory. The *head dependency graph* of T is the directed graph such that

- its vertices are the literals of σ, and
- for each rule $l_1 \vee \cdots \vee l_n \Leftarrow F$ of T, it has an edge from each l_i to each $\overline{l_j}$ where $j \neq i$, $1 \leq i, j \leq n$.[3]

[3] $\overline{l}$ denotes the literal complementary to literal l.

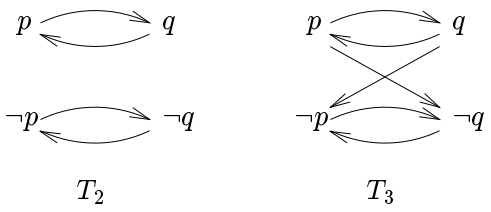

Figure 10.2: The head dependency graphs of T_2, T_3

Thus head dependency graphs of causal theories differ from positive dependency graphs of logic programs in two ways. First their vertices are literals, and not only atoms. Second, the edges of a graph come from the heads of the rules.

For instance, the head dependency graphs of T_2 and T_3 (Section 3.3) are shown in Figure 10.2.1.

Similarly to logic programs, a nonempty consistent set L of literals is called a *loop* of T if, for every pair l_1, l_2 of literals in L, there exists a path from l_1 to l_2 in the head dependency graph of T such that all vertices in this path belong to L. A loop is called *trivial* if the loop consists of a single literal such that there is no edge from the literal to itself in the head dependency graph.

For instance, T_2 has six loops: $\{p\}$, $\{\neg p\}$, $\{q\}$, $\{\neg q\}$, $\{p, q\}$, $\{\neg p, \neg q\}$, among which the first four are trivial.

The following fact easily follows from the definition of a loop. Given a set L of literals, $\overline{L}$ is the set of literals complementary to literals in L.

Fact 1 *For any canonical causal theory T, if a set L of literals is a loop of T, $\overline{L}$ is a loop of T also.*

For any set Y of atoms, the *external support formula* for Y is the disjunction

of the conjunctions

$$G \wedge \bigwedge_{l \in F \setminus Y} \overline{l}$$

for all rules $F \Leftarrow G$ of T such that

- $F \cap Y \neq \emptyset$, and
- $F \cap \overline{Y} = \emptyset$.

We denote the external support formula by $ES_{T,Y}$.

Given a canonical causal theory T, propositional theory $Tr_b(T)$ consists of

(i) the implications

$$G \supset F$$

for all rules $F \Leftarrow G$ in T,

(ii) the implications

$$\bigvee Y \ \supset \ ES_{T,Y} \tag{10.3}$$

for all consistent sets Y of literals of σ.

Propositional theory $Tr_c(T)$ is defined in the same way except that instead of formulas (10.3),

- theory $Tr_c(T)$ includes

$$\bigwedge L \ \supset \ ES_{T,L} \tag{10.4}$$

for all loops L of T.

When L is a loop, we call formula (10.4) the *conjunctive loop formula* of L for T.

Theorem 2 *For any canonical causal theory T and any interpretation X of the signature of T, the following conditions are equivalent to each other:*

(*a*) *X is a model of T.*

(*b*) *X is a model of* $Tr_b(T)$.

(*c*) *X is a model of* $Tr_c(T)$.

Since conditions (*b*) and (*c*) of Theorem 2 are equivalent to each other, any intermediate condition between the two is also equivalent to (*a*)–(*c*). In particular we consider the following translations, Tr_d and Tr_e, which are defined in the same way as Tr_b except that instead of formulas (10.3),

- theory $Tr_d(T)$ includes

$$\bigwedge Y \supset ES_{T,Y}$$

  for all nonempty consistent sets Y of literals of σ.[4]

- theory $Tr_e(T)$ includes

$$\bigvee L \supset ES_{T,L} \tag{10.5}$$

  for all loops L of T.

When L is a loop, we call formula (10.5) the *disjunctive loop formula* of L for T.

Corollary 5 *For any canonical causal theory T and any interpretation X of the signature of T, the following conditions are equivalent to each of conditions* (*a*)–(*c*) *of Theorem 2:*

(*d*) *X is a model of* $Tr_d(T)$.

(*e*) *X is a model of* $Tr_e(T)$.

[4]The requirement that Y be consistent can be dropped since the implication is trivially true in this case.

Notice that the set of formulas in each of the conditions (b)–(e) consists of two parts. The first is a modular translation of T into propositional logic: it reads each rule of T as an implication. On the other hand, the second part is a non-modular translation. For instance, to find loops and to write the consequents of (10.4), one has to look at the whole theory.

For example, consider T_2 from Section 3.3. The theory is not definite, so that we cannot use the literal completion method to find its models. But condition (c) of Theorem 2 tells us that the models of T_2 are exactly the models of

$$(p \vee \neg q) \wedge (\neg p \vee q)$$
$$\wedge (p \supset q) \wedge (q \supset p) \wedge (\neg p \supset \neg q) \wedge (\neg q \supset \neg p) \wedge (p \wedge q \supset \bot) \wedge (\neg p \wedge \neg q \supset \bot).$$

Theory T_3 from Section 3.3 is another example to which we cannot apply literal completion. According to Theorem 2, the models of T_3 are exactly the models of

$$(p \vee \neg q) \wedge (\neg p \vee q) \wedge (p \vee q) \wedge (p \supset q \vee \neg q) \wedge (q \supset p \vee \neg p)$$
$$\wedge (\neg p \supset \neg q) \wedge (\neg q \supset \neg p) \wedge (p \wedge q \supset \top) \wedge (\neg p \wedge \neg q \supset \bot).$$

Note that all translations (b)—(e) involve the exponential number of loops in the worst case, and this may be seen as a defect of the translations. However, assuming a conjecture from the theory of computational complexity which is widely believed to be true, we can show that any equivalent transformation from causal theories to propositional formulas without introducing new atoms involves a significant increase in size, in the worst case. This is a consequence of a similar result for logic programs proved by Lifschitz and Razborov [2004], in combination with the lemma from [Giunchiglia *et al.*, 2004, Section 8.3].

10.2.2 Completion and Tight Causal Theories

We can extend the literal completion method to canonical theories, not necessarily definite, as follows. The *(literal) completion* of a canonical theory T, denoted by $Comp(T)$, is the conjunction of T with the implications

$$l \supset ES_{T,\{l\}}$$

for all literals l of the signature of T. Note that this definition is a generalization of the literal completion of definite theories (Section 3.4).

Since every singleton set of literals is a (trivial) loop, $Comp(T)$ can be viewed as the conjunction of T with loop formulas of all trivial loops of T. Thus it is clear that $Tr_c(T)$ implies $Comp(T)$, but not vice versa. The following is a corollary to Theorem 2 (Section 10.2.1):

Corollary 6 *For any canonical theory T, if X is a model of T then it is a model of $Comp(T)$.*

As in the case of logic programs (see Section 3.1), we can define a "tight" causal theory, for which the implication in the other direction also holds. Our definition is based on the notion of a loop. For a canonical causal theory T, we will say that T is *tight* if all loops of T are trivial. It is clear that any definite theory is tight. On the other hand, theories T_2 and T_3 (Section 3.3) are not tight. The following example is a nondefinite theory which is tight:

$$\begin{aligned} p \vee \neg q &\Leftarrow \top \\ r \vee q &\Leftarrow \top \\ \neg r &\Leftarrow \neg r. \end{aligned}$$

Since $Comp(T)$ and $Tr_c(T)$ are the same if T is tight, we get the following corollary to Theorem 2, which generalizes the proposition from Section 3.4.

Corollary 7 *For any tight canonical causal theory T and any interpretation X of the signature of T, X is a model of T iff X is a model of $Comp(T)$.*

For tight theories, the translations from the previous section give polynomial-size propositional formulas.

10.2.3 Turning Nondefinite Theories into Definite Theories

A corollary of Theorem 2 tells us that any nondefinite theory can be turned into an equivalent definite theory. Let T be a canonical causal theory of a signature σ. For every rule $F \Leftarrow G$ of T, the corresponding set of definite rules $DR(F \Leftarrow G)$ is defined as follows:

$$DR(F \Leftarrow G) = \left\{ l \Leftarrow G \wedge \bigwedge_{l' \in F \setminus \{l\}} \overline{l'} \ : \ l \in F \right\}$$

if $|F| > 1$; $DR(F \Leftarrow G) = \{F \Leftarrow G\}$ otherwise.

The set of definite rules corresponding to T is the union of $DR(r)$ for all rules r in T:

$$DR(T) = \bigcup_{r \in T} DR(r).$$

Note that $DR(T) = T$ when T is definite.

For example, for T_3 from Section 3.3, $DR(T_3)$ is

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

and its only model is $\{p, q\}$, which is the only model of T_3 also.

For T_2 from Section 3.3, which has no models, $DR(T_2)$ is

$$\begin{aligned} p &\Leftarrow q \\ \neg q &\Leftarrow \neg p \\ \neg p &\Leftarrow \neg q \\ q &\Leftarrow p, \end{aligned} \tag{10.6}$$

which has two models, $\{p, q\}$ and $\{\neg p, \neg q\}$.

According to Proposition 5 from [Giunchiglia *et al.*, 2004], adding a constraint $\bot \Leftarrow F$ (Section 3.3) to a causal theory T does not introduce new models, but simply eliminates the models of T that does not satisfy F. Thus we get the following as a corollary to Theorem 2.

Corollary 8 *For any canonical causal theory T, the following conditions are equivalent to each other:*

(a) X is a model of T.

(b) X is a model of $DR(T) \cup \{\bot \Leftarrow \neg(\bigvee Y \supset ES_{T,Y}) : Y$ is a consistent set of literals of $\sigma\}$.

(c) X is a model of $DR(T) \cup \{\bot \Leftarrow \neg(\bigwedge L \supset ES_{T,L}) : L$ is a nontrivial loop of $T\}$.

(d) X is a model of $DR(T)$

$\cup \{\bot \Leftarrow \neg(\bigwedge Y \supset ES_{T,Y}) : Y$ is a nonempty consistent set of literals of $\sigma\}$.

(e) X is a model of $DR(T) \cup \{\bot \Leftarrow \neg(\bigvee L \supset ES_{T,L}) : L$ is a nontrivial loop of $T\}$.

For example, the constraints that express the conjunctive loop formulas of nontrivial loops for T_2 are

$$\begin{aligned} \bot &\Leftarrow \neg(p \wedge q \supset \bot) \\ \bot &\Leftarrow \neg(\neg p \wedge \neg q \supset \bot). \end{aligned} \tag{10.7}$$

Corollary 8 tells us that the models of T_2 are exactly the models of the causal theory which consists of (10.6) and (10.7).

As another example, consider the action description shown in Figure 5.1. Recall that it is nondefinite because of the last causal law, which is translated into a set of causal rules:

$$i\!:\mathit{Turning}(1) \equiv\ i\!:\mathit{Turning}(2) \Leftarrow i\!:\mathit{Connected}. \tag{10.8}$$

According to Corollary 8, the causal theory corresponding to Figure 5.1 can be turned into a definite theory with the same set of models by replacing the rule (10.8) with the following rules [5]:

$$\begin{aligned} i\!:\neg\mathit{Turning}(x) &\Leftarrow i\!:\neg\mathit{Turning}(x_1) \wedge i\!:\mathit{Connected} \qquad (x \neq x_1) \\ i\!:\mathit{Turning}(x) &\Leftarrow i\!:\mathit{Turning}(x_1) \wedge i\!:\mathit{Connected} \qquad (x \neq x_1) \\ \bot &\Leftarrow \neg\big(\textstyle\bigwedge_x i\!:\mathit{Turning}(x) \supset \bigvee_x i\!:\mathit{MotorOn}(x)\big). \end{aligned} \tag{10.9}$$

[5] The first rule can be dropped without changing the set of models.

10.2.4 Transitive Closure

The comparison of the definition of a loop in logic programs and in causal logic can guide us in translating a representation from one formalism to the other. For example, the dependency graph of $p \leftarrow q$ and the dependency graph of $p \Leftarrow q$ are different: while the former has an edge from p to q, the latter has no edges. On the other hand, $q \supset p \Leftarrow \top$ has two edges: one from p to q, and the other from $\neg q$ to $\neg p$.

In logic programming the following set of rules describes the transitive closure tc of a binary relation p on a set A:

$$\begin{aligned} & p(x,y) && \textit{for any pair } x,y \in A \textit{ such that } p(x,y) \textit{ holds} \\ & tc(x,y) \leftarrow p(x,y) \\ & tc(x,z) \leftarrow p(x,y), tc(y,z). \end{aligned} \tag{10.10}$$

One might be tempted to write the corresponding representation in causal logic as follows:

$$\begin{aligned} p(x,y) &\Leftarrow \top && \textit{for any pair } x,y \in A \textit{ such that } p(x,y) \textit{ holds} \\ tc(x,y) &\Leftarrow p(x,y) \\ tc(x,z) &\Leftarrow p(x,y) \wedge tc(y,z) \\ \neg p(x,y) &\Leftarrow \neg p(x,y) \\ \neg tc(x,y) &\Leftarrow \neg tc(x,y). \end{aligned} \tag{10.11}$$

Note that the completion of (10.10) is equivalent to the completion of (10.11). If p is acyclic, then tc in (10.11) describes the transitive closure correctly. Otherwise, the representation may allow spurious models that do not correspond to the transitive closure.

The presence of the spurious models is related to the "cyclic causality" in the third rule of (10.11). The loop formulas for (10.11) are not equivalent to the loop formulas for (10.10). In (10.10) the third rule tells us that the positive dependency graph has edges that go from $tc(x,z)$ to $tc(y,z)$, while in (10.11) the corresponding rule does not contribute to the edges of the head dependency graph. Indeed, (10.11) has trivial loops only.

This problem can be corrected by moving $tc(y,z)$ in the third rule from the body to the head, to ensure that the head dependency graph contains the corresponding edges:

$$tc(y,z) \supset tc(x,z) \Leftarrow p(x,y).^{6}$$

The modified causal theory may have more loops than (10.10), but the loop formulas for these extra loops are tautologies, because each of the loops contains at least one negative literal, and there is a rule $\neg c \Leftarrow \neg c$ in the theory for every atom c. Thus it is easy to see that the loop formulas for the modified causal theory are equivalent to the loop formulas for (10.10). The translation of the causal logic representation of transitive closure to the corresponding logic program provides an alternative proof of Theorem 2 from [Doğandağ *et al.*, 2004], which shows the correctness of the modified casual theory for representing transitive closure. According to Corollary 8, *tc* can also be described by definite theories using the translation from Section 10.2.3.

[6] According to Proposition 2 of [Lee, 2004], we can also write $p(x,y) \wedge tc(y,z) \supset tc(x,z) \Leftarrow \top$. Since $p(x,y)$ does not contribute to any loops, moving $p(x,y)$ from the head to the body does not change loop formulas. The case is similar with the second rule of (10.11).

10.3 Loop Formulas for Arbitrary Causal Theories

We can extend Theorem 2 to arbitrary causal theories, not necessarily canonical. An example of a non-canonical causal theory is given in Section 10.2.3; the theory is non-canonical because some rules have equivalences in the heads.

About an occurrence of a literal l in a formula, we say that it is *singular* if l is a positive literal preceded by $\neg$, and that it is *regular* otherwise. Given a formula F, $NNF(F)$ denotes the *negation normal form* of F, that is, the formula obtained from F by distributing $\neg$ over $\wedge$ and $\vee$ until it applies to atoms only.

Let T be a causal theory of a signature σ. The *head dependency graph* of T is the directed graph such that

- its vertices are the literals of σ, and
- it includes an edge from a vertex l to a vertex l' if there is a rule $F \Leftarrow G$ in T such that l occurs regularly in $NNF(F)$, and $\overline{l'}$ occurs regularly in $NNF(F)$.

This definition reduces to the earlier definition (Section 10.2.1) when T is canonical.

Given a formula F and a consistent set Y of literals, by F_Y we denote the formula obtained from F by replacing

- each occurrence of atom a such that $a \in Y$ by $\bot$, and
- each occurrence of atom a such that $\neg a \in Y$ by $\top$.

By T_Y we denote the theory obtained from T by replacing all rules $F \Leftarrow G$ in T with $F_Y \Leftarrow G$. Note that in the process of constructing T_Y we transform only the heads of the rules.

In application to canonical causal theories, this operation is closely related to external support formula:

Proposition 14 *Let T be a canonical causal theory of a signature σ, and X an interpretation of σ that satisfies T. For any consistent set Y of literals of σ, X satisfies $ES_{T,Y}$ iff X does not satisfy T_Y.*

The translations Tr_b, Tr_c, Tr_d, Tr_e from Section 10.2 can be extended to arbitrary causal theories as follows: propositional theory $Tr_b(T)$ consists of

(i) the implications

$$G \supset F$$

for all rules $F \Leftarrow G$ in T, and

(ii) the implications

$$\bigvee Y \ \supset \ \neg T_Y \tag{10.12}$$

for all consistent sets Y of literals of σ.

In the case when T is canonical, this definition differs from the definition from Section 10.2.1 only in that in formulas (10.12) we use T_Y instead of $ES_{T,Y}$. In view of Proposition 14 this difference is not essential.

The translations Tr_c, Tr_d, and Tr_e are defined in the same way except that instead of formulas (10.12),

- theory $Tr_c(T)$ includes

$$\bigwedge L \ \supset \ \neg T_L \tag{10.13}$$

for all loops L of T,

- theory $Tr_d(T)$ includes

$$\bigwedge Y \supset \neg T_Y$$

for all nonempty consistent sets Y of literals of σ,

- theory $Tr_e(T)$ includes

$$\bigvee L \supset \neg T_L \tag{10.14}$$

for all loops L of T.

Theorem 3 *For any causal theory T and any interpretation X of the signature of T, the following conditions are equivalent to each other:*

(a) X is a model of T.

(b) X is a model of $Tr_b(T)$.

(c) X is a model of $Tr_c(T)$.

Corollary 9 *For any causal theory T and any interpretation X of the signature of T, the following conditions are equivalent to each of conditions (a)–(c) of Theorem 3:*

(d) X is a model of $Tr_d(T)$.

(e) X is a model of $Tr_e(T)$.

The idea above can also be used to prove theorems about the relationship between logic programs and causal logic. For instance, the proof of Proposition 2 from [Lee, 2004], which shows how to embed disjunctive logic programs into causal logic, is given by turning both logic programs and causal theories into propositional theories and showing that these propositional theories are equivalent to each other.

10.4 Proofs

In this section, we prove Proposition 14 and Theorem 3. These two theorems imply Theorem 2.

We will sometimes identify a causal theory T with the corresponding propositional theory, and say that an interpretation *satisfies* T if it satisfies the propositional theory.

10.4.1 Proof of Proposition 14

Proposition 14 *Let T be a canonical causal theory of a signature σ, and X an interpretation of σ that satisfies T. For any consistent set Y of literals of σ, X satisfies $ES_{T,Y}$ iff X does not satisfy T_Y.*

Proof (*Left-to-right*) Assume that $X \models ES_{T,Y}$. Then there is a rule

$$F \Leftarrow G \tag{10.15}$$

in T such that

$$\begin{aligned} &X \models G, \\ &X \cap (F \setminus Y) = \emptyset \\ &F \cap Y \neq \emptyset, \text{ and} \\ &F \cap \overline{Y} = \emptyset. \end{aligned} \tag{10.16}$$

It follows that $X \not\models F_Y$, and consequently, $X \not\models T_Y$.

(*Right-to-left*) Assume that $X \not\models T_Y$. We first show that X satisfies $F_Y \Leftarrow G$ for every rule $F \Leftarrow G$ that does not satisfy (10.16).

- For every rule $F \Leftarrow G$ such that $X \not\models G$, $X \cap (F \setminus Y) \neq \emptyset$, or $F \cap \overline{Y} \neq \emptyset$, X satisfies $F_Y \Leftarrow G$ trivially.

- For every rule $F \Leftarrow G$ such that $F \cap Y = \emptyset$, since X satisfies T, it follows that $X \not\models G$, or $X \cap (F \setminus Y) \neq \emptyset$. In either case, it is easy to check that X satisfies $F_Y \Leftarrow G$.

It follows that there exists a rule that satisfies (10.16). Therefore $X \models ES_{T,Y}$. ∎

10.4.2 Proof of Theorem 3

The proof of Theorem 3 uses the following lemma, proved in Section 10.4.3.

Main Lemma *Let T be a causal theory of a signature σ, X an interpretation of σ that satisfies T, and Y a nonempty consistent set of literals of σ. If X does not satisfy T_L for any loop L that is contained in Y, then X does not satisfy T_Y.*

The proof of Theorem 3 uses the following facts as well.

Fact 2 *For any causal theory T and any interpretation X of the signature of T, $X \models T$ iff $X \models T^X$.*

This is immediate by structural induction.

Fact 3 *Let F be a formula, T a causal theory, X an interpretation of the signature of T, and Y a consistent set of literals.*

(*i*) $X \models F_Y$ *iff* $(X \setminus Y) \cup \overline{Y} \models F$.

(*ii*) $X \models T_Y$ *iff* $(X \setminus Y) \cup \overline{Y} \models T^X$.

Part (*i*) is immediate by structural induction. Part (*ii*) follows from (*i*).

Theorem 3 *For any causal theory T and any interpretation X of the signature of T, the following conditions are equivalent to each other:*

(*a*) *X is a model of T.*

(*b*) *X is a model of* $Tr_b(T)$.

(*c*) *X is a model of* $Tr_c(T)$.

Proof *From* (*b*) *to* (*c*) is clear.

From (*a*) *to* (*b*): Let X be a model of T. From the definition of a model, it follows that X satisfies T. Let Y be any consistent set of literals such that $Y \cap X \neq \emptyset$. Since X is a model of T, $(X \setminus Y) \cup \overline{Y}$, which is different from X, does not satisfy T^X. By Fact 3, it follows that X does not satisfy T_Y.

From (*c*) *to* (*a*): Assume that X satisfies T, and, for every loop L of T that is contained in X, X does not satisfy T_L. First, by Fact 2, $X \models T^X$. Let Y be any interpretation that is different from X. We will show that $Y \not\models T^X$. Let $Z = X \setminus Y$. Since Z is nonempty, and $X \not\models T_L$ for any loop L that is contained in Z, by the main lemma, it follows that $X \not\models T_Z$, which is equivalent to $(X \setminus Z) \cup \overline{Z} \not\models T^X$, i.e., $Y \not\models T^X$ by Fact 2. Therefore X is the unique interpretation satisfying T^X. ∎

10.4.3 Proof of the Main Lemma

Lemma 3 *Let T be a causal theory of a signature* σ, *X an interpretation of* σ *that satisfies T, Y a consistent set of literals of* σ, *and L a nonempty subset of Y such that the head dependency graph of T has no edge from a literal in L to a literal in* $Y \setminus L$. *If X does not satisfy* T_L, *then X does not satisfy* T_Y.

Proof Assume that X does not satisfy T_L. There exists a rule

$$H \leftarrow B$$

in T such that X satisfies B, but does not satisfy H_L. By Fact 3,

$$(X \setminus L) \cup \overline{L} \not\models H. \tag{10.17}$$

On the other hand,

$$X \models H \tag{10.18}$$

because X satisfies T. From (10.17) and (10.18), it follows that at least one literal in L occurs regularly in $NNF(H)$.

Next we will show that $X \not\models T_Y$. Since the head dependency graph of T has no edge from a literal in L to a literal in $Y \setminus L$, it follows that there is no literal $l \in Y \setminus L$ such that $\overline{l}$ occurs regularly in $NNF(H)$. It follows from (10.17) that

$$(X \setminus Y) \cup \overline{Y} \not\models H,$$

which is equivalent to $X \not\models H_Y$ by Fact 3. Therefore, $X \not\models T_Y$. ∎

Main Lemma *Let T be a causal theory of a signature σ, X an interpretation of σ that satisfies T, and Y a nonempty consistent set of literals of σ. If X does not satisfy T_L for any loop L that is contained in Y, then X does not satisfy T_Y.*

Proof In view of Lemma 3, it is sufficient to show that there exists a loop L such that the head dependency graph of T has no edge from a literal in L to a literal in $Y \setminus L$. We will show the existence of such loops.

Let G be the subgraph of the head dependency graph of T induced by Y, and let G' be the graph obtained from G by collapsing the strongly connected components of G (that is, the vertices of G' are the strongly connected components of G and G' has an edge from vertex V to vertex V' if G has an edge from a literal

in V to a literal in V'). Since Y is nonempty, there is at least one loop in Y. Consequently, there is at least one vertex in G'.

It follows that there exists a terminal vertex in G'. Let L be that vertex. It is clear that there is no edge from a literal in L to a literal in $Y \setminus L$ in the head dependency graph of T. ∎

Chapter 11

Splitting Causal Theories

The splitting set theorem [Lifschitz and Turner, 1994] allows us, under certain conditions, to split a logic program into two parts and determine how the answer sets of the first part are affected by adding the second part. In this chapter, we extend this idea to causal logic. The proof of the theorem uses Theorem 3 from the previous chapter. In Section 11.2, we illustrate the usefulness of the splitting set theorem by using it to prove a proposition regarding the use of statically determined fluents in $\mathcal{C}+$.

11.1 Splitting Set Theorem for Causal Logic

Let us consider T_1 (Section 3.3). Without the first rule, the theory consists of the rules of the signature $\{q\}$, that is,

$$\begin{aligned} q &\Leftarrow q \\ \neg q &\Leftarrow \neg q. \end{aligned} \tag{11.1}$$

This theory, assuming its signature is $\{q\}$, has two models: $\{q\}$, $\{\neg q\}$. If we select $\{q\}$, and "plug in the value" to the first rule, then the resulting theory consists of a rule of the signature $\{p\}$, that is

$$p \Leftarrow \top. \tag{11.2}$$

This theory, assuming its signature is $\{p\}$, has one model: $\{p\}$. The union of the two models, $\{q\} \cup \{p\}$, coincides with the model of T_1.

On the other hand, if we select $\{\neg q\}$, the theory is

$$p \Leftarrow \bot, \tag{11.3}$$

which has no models.

The example gives us an idea how a causal theory can be "split." More precisely, we have the following result, which is similar to the splitting set theorem for logic programs [Lifschitz and Turner, 1994].

Let T be a causal theory of a signature σ. A subset U of σ is a *splitting set* for T if, for every rule $F \Leftarrow G$ in T such that F contains a constant from U, all constants occurring in F or G belong to U also. The *bottom* of T relative to a splitting set U, denoted by $b_U(T)$, is the causal theory of the signature U consisting of all rules $F \Leftarrow G$ from T such that all constants occurring in F or G belong to U. By $t_U(T)$ (the *top* of T relative to U) we denote the causal theory of the signature σ that consists of all rules of T not included in $b_U(T)$.

For example, $\{q\}$ is a splitting set for T_1; the bottom of T_1 relative to $\{q\}$, $b_{\{q\}}(T_1)$, is theory (11.1) with the signature $\{q\}$; the top of T_1 relative to $\{q\}$, $t_{\{q\}}(T_1)$, consists of the first rule of T_1 with the signature $\{p\}$.

Let F be a formula of the signature σ, U a subset of σ, and X a set of atoms $c = v$ such that $c \in U$. By $e_U(F, X)$ we denote the formula obtained from F by

replacing each occurrence of atom $c=v$ with $c \in U$ by $\top$ if it belongs to X and by $\bot$ otherwise. For a causal theory T of the signature σ, by $e_U(T, X)$ we denote the causal theory of the signature $\sigma \setminus U$ that consists of the rules $e_U(F, X) \Leftarrow e_U(G, X)$ for all rules $F \Leftarrow G$ of T.

For example, the theory $e_{\{q\}}(t_{\{q\}}(T_1, \{q\})$ is (11.2); the theory $e_{\{q\}}(t_{\{q\}}(T_1, \{\neg q\})$ is (11.3).

Theorem 4 *Let T be a causal theory whose signature is σ, and U a splitting set for T. An interpretation of σ is a model of T iff it can be written as $X \cup Y$ where X is a model of $b_U(T)$ and Y is a model of $e_U(t_U(T), X)$.*

According to Theorem 4, the model of T_1 can be written as $\{q\} \cup \{p\}$ where $\{q\}$ is a model of $b_{\{q\}}(T_1)$ and $\{p\}$ is a model of $e_{\{q\}}(t_{\{q\}}(T_1), \{q\})$.

11.2 Proof of Proposition 4

Proposition 4 *Let D be an action description whose signature is σ, Q a set of statically determined fluent constants such that $\sigma \cap Q = \emptyset$, and D_Q an action description which consists of causal laws of the form*

$$\textbf{caused } q \textbf{ if } F$$

where $q \in Q$ and F is a formula of σ, and the causal laws

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

for all $q \in Q$. Then the transition system of $D \cup D_Q$ is isomorphic to the transition system of D.

Proof First, take the signature $U = 0:\sigma$ as a splitting set of $(D \cup D_Q)_0$. Then $b_U((D \cup D_Q)_0)$ is D_0 and $t_U((D \cup D_Q)_0)$ is $(D_Q)_0$. By Theorem 4, any model of $(D \cup D_Q)_0$ can be written as $X \cup Y$ where X is a model of D_0 and Y is a model of $e_U((D_Q)_0, X)$. Since $\sigma \cap Q = \emptyset$, it is easy to check that given X, $e_U((D_Q)_0, X)$ has a unique model of the signature $0:Q$ (consider the completion of $e_U((D \cup D_Q)_0, X)$). Thus it follows that there is a 1–1 correspondence between the states of D and the states of $D \cup D_Q$.

Now take the signature $U = 0:\sigma \cup 1:\sigma$ as a splitting set of $(D \cup D_Q)_1$. Then $b_U((D \cup D_Q)_1)$ is D_1 and $t_U((D \cup D_Q)_1)$ is $(D_Q)_1$. By Theorem 4, any model of $(D \cup D_Q)_1$ can be written as $X \cup Y$ where X is a model of D_1 and Y is a model of $e_U((D_Q)_1, X)$. Since $\sigma \cap Q = \emptyset$, it is easy to check that given X, $e_U((D_Q)_1, X)$ has a unique model of the signature $0:Q \cup 1:Q$ (again, consider the completion of $e_U((D \cup D_Q)_1, X)$). Thus it follows that there is a 1–1 correspondence between the transitions of D and the transitions of $D \cup D_Q$. ∎

11.3 Related Work

The splitting set theorem presented in this chapter is closely related to is the splitting set theorem for default logic presented in [Turner, 1996]. Since a Boolean causal theory can be viewed as a default theory in the sense of [Reiter, 1980] by identifying a causal rule $F \Leftarrow G$ with the default

$$\frac{: G}{F}$$

[Giunchiglia *et al.*, 2004, Proposition 10], one can also derive a splitting set theorem for causal logic from Turner's theorem. However, a straightforward derivation would

require that each body of a rule be either a formula of the signature U or the signature $\sigma \setminus U$, which our splitting set theorem does not require. Moreover, our theorem is not limited to Boolean theories.

11.4 Proof of the Splitting Set Theorem

Lemma 4 *For any causal theory T whose signature is σ, and any splitting set U for T, every loop L of T is either a loop of $b_U(T)$ or a loop of $t_U(T)$ over $\sigma \setminus U$.*

Proof Easily follows from the fact that the head dependency graph of T has no edge from a literal of U to a literal of $\sigma \setminus U$. ∎

Lemma 5 *Let T be a causal theory whose signature is σ, U a splitting set for T, and X an interpretation of σ that satisfies T.*

(i) For every loop L of $b_U(T)$,

$$X \models b_U(T)_L \text{ iff } X \models T_L.$$

(ii) For every loop L of $T \setminus b_U(T)$ over $\sigma \setminus U$,

$$X \models (T \setminus b_U(T))_L \text{ iff } X \models T_L.$$

Proof Follows from Lemma 4. ∎

Lemma 6 *Let T be a causal theory whose signature is σ, U a splitting set for T, and X an interpretation of U. For any loop L of $t_U(T)$ over $\sigma \setminus U$,*

$$e_U(t_U(T), X)_L = e_U(t_U(T)_L, X).$$

Proof Clear from the definitions of e_U and T_L. ∎

Theorem 4 *Let T be a causal theory whose signature is σ, and U a splitting set for T. An interpretation of σ is a model of T iff it can be written as $X \cup Y$ where X is a model of $b_U(T)$ and Y is a model of $e_U(t_U(T), X)$.*

Proof (*Left-to-right*) Assume that Z is a model of T. We will show that $Z = X \cup Y$ for two sets X and Y of literals such that X is a model of $b_U(T)$, and Y is a model of $e_U(t_U(T), X)$. Take X to be the set of literals of U that belong to Z and Y to be the set of literals of $\sigma \setminus U$ that belong to Z. It is clear that $Z = X \cup Y$. By Theorem 3, $Z \models T$ and $Z \not\models T_L$ for every loop L of T.

First we will show that X is a model of $b_U(T)$. Since $Z \models T$ and the signature of $b_U(T)$ is U, it follows that

$$X \models b_U(T).$$

Since $Z \not\models T_L$ for every loop L of $b_U(T)$, by Lemma 5 (i), it follows that

$$X \not\models b_U(T)_L$$

for every loop L of $b_U(T)$. Therefore, by Theorem 3, X is a model of $b_U(T)$.

Next we will show that Y is a model of $e_U(t_U(T), X)$. Since $X \cup Y \models t_U(T)$, by the definition of e_U,

$$Y \models e_U(t_U(T), X).$$

On the other hand, since $Z \not\models T_L$ for every loop L of T, by Lemma 5 (ii),

$$X \cup Y \not\models t_U(T)_L$$

for every loop L of $t_U(T)$ over $\sigma \setminus U$. By the definition of e_U,

$$Y \not\models e_U(t_U(T)_L, X),$$

or by Lemma 6,

$$Y \not\models e_U(t_U(T), X)_L,$$

for the same loops. Therefore, by Theorem 3, Y is a model of $e_U(t_U(T), X)$.

(*Right-to-left*) Assume that X is a model of $b_U(T)$ and Y is a model of $e_U(t_U(T), X)$. Then we need to show that $X \cup Y$ is a model of T.

It is clear that X satisfies $b_U(T)$. From $Y \models e_U(t_U(T), X)$, it holds that

$$X \cup Y \models t_U(T).$$

Therefore $X \cup Y \models b_U(T) \cup t_U(T) = T$.

Next we will show that $X \cup Y \not\models T_L$ for every loop L of T. Take any loop L of T. By lemma 4, L is a loop of either $b_U(T)$ or a loop of $t_U(T)$ over $\sigma \setminus U$.

- If L is a loop of $b_U(T)$, then by Theorem 3, $X \not\models b_U(T)_L$. By Lemma 5 (i), $X \not\models T_L$.

- If L is a loop of $t_U(T)$ over $\sigma \setminus U$, then by Theorem 3,

  $$Y \not\models e_U(t_U(T), X)_L,$$

  and by Lemma 6,

  $$Y \not\models e_U(t_U(T)_L, X).$$

  By the definition of e_U, it follows that

  $$X \cup Y \not\models t_U(T)_L,$$

  and by Lemma 5 (ii),

  $$X \cup Y \not\models T_L.$$

Therefore Z is a model of T by Theorem 3. ∎

Chapter 12

Conclusion

12.1 Summary of Contributions

The main contributions of this dissertation are as follows.

- We identified several essential limitations of the McCain–Turner causal logic and action language $\mathcal{C}$ in application to describing commonsense knowledge about actions, and proposed multi-valued causal logic and language $\mathcal{C}+$, which overcome these limitations. Language $\mathcal{C}+$, a high level notation for multi-valued causal theories, can represent non-propositional fluents, defined fluents, rigid constants, and defeasible causal laws. Despite many advanced features, it has a simple and elegant formal semantics.

- We redesigned and reimplemented CCALC to account for these extensions. The input language of the new CCALC provides a convenient and concise syntax for writing action descriptions in the definite fragment of $\mathcal{C}+$.

- We identified the concept of an additive fluent and proposed a method for

describing additive fluents in $\mathcal{C}+$. We extended CCALC accordingly to cover additive fluents, and applied it to several examples of commonsense reasoning.

- We tested expressive possibilities of $\mathcal{C}+$ by formalizing action domains of non-trivial size, much more complicated than "toy problems."

- By formalizing McCarthy's elaborations of the Missionaries and Cannibals Puzzle, we showed that, to a certain degree, the goal of elaboration tolerance is met by the input language of the extended CCALC. Each enhancement was obtained from the basic formalization by simply adding more postulates.

- We showed how to turn an arbitrary causal theory, not necessarily definite, into a set of formulas in propositional logic using the concept of a loop formula. As a corollary we showed that any nondefinite theory can be turned into a definite theory. The result provides a way to extend CCALC to deal with arbitrary causal theories.

12.2 Topics for Future Work

The following is a list of topics for future work that would improve upon the results of this dissertation.

- We have not yet considered how to incorporate sensing actions—actions that affect the agent's knowledge but have no effect on the world, and we have not shown how to incorporate probabilistic reasoning in $\mathcal{C}+$.

- Used as a planner, CCALC does not rely on domain-specific control knowledge. It has been noted that declarative control knowledge sometimes drastically

improves the performance of planners. Although CCALC can solve prediction and postdiction problems with incomplete information, it does not handle "conformant planning"—generating plans that are guaranteed to succeed with incomplete initial conditions.

- CCALC has mainly been a research tool used to test the expressiveness of its input language. For CCALC to be used as a more practical system, the implementation should consider efficiency and optimization more seriously.

- The current version of CCALC does not operate with real numbers, and even integer arithmetic is implemented in a way that becomes inefficient when large integers are needed. It may be possible to lift these limitations by developing an interface with search engines other than SAT solvers, such as those based on linear programming, as in [Wolfman and Weld, 1999], or on integer programming, as in [Kautz and Walser, 1999].

- Our translation of an arbitrary causal theory into formulas in propositional logic (or into a definite theory) can be used to extend CCALC to handle nondefinite theories. We may be able to identify a useful subclass of causal theories, which is more general than the class of definite theories but still can be computed efficiently. One such extension was proposed in [Doğandağ *et al.*, 2004]. Also there is a need to better understand how loops can be computed.

- CCALC may serve as a general-purpose reasoning tool which is far more expressive than many other reasoning systems. For instance, CCALC may be used to formalize the behavior of software/hardware systems, which would allow "deep reasoning" about their behavior. This will lead to many interesting

applications such as online help systems, for which an elaboration tolerant formalism can be useful for maintaining knowledge bases. CCALC may be applied to formalizing and testing *workflows*, a series of tasks performed by various cooperative and coordinated agents to achieve a desired goal.

Appendix A

Solutions for Elaborations of MCP found by CCalc

A.1 Solution for the Basic Problem

```
| ?- loadf 'basic-test'.
% loading file /v/filer3/v2q021/appsmurf/ccalc/macros.std
% loading file /v/filer3/v2q021/appsmurf/mcp/basic-test
% loading file /v/filer3/v2q021/appsmurf/ccalc/additive
% loading file /v/filer3/v2q021/appsmurf/ccalc/arithmetic
% loading file /v/filer3/v2q021/appsmurf/mcp/basic
% loading file /v/filer3/v2q021/appsmurf/mcp/common2
% loading file /v/filer3/v2q021/appsmurf/mcp/common1
% in transition mode...
% 130 atoms, 246 rules, 809 clauses (92 new atoms)
% Grounding time: 3.53 seconds
% Completion time: 0.56 seconds
% Total time: 4.09 seconds

yes
| ?- query 0.
```

```
% Shifting atoms and clauses... done. (0.02 seconds)
% After shifting: 2004 atoms (including new atoms), 7685 clauses
% Writing input clauses... done.  (0.56 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 1.11 seconds (prep 0.09 seconds, search 1.02 seconds)

No solution with maxstep 10.

% Shifting atoms and clauses... done. (0.03 seconds)
% After shifting: 2202 atoms (including new atoms), 8449 clauses
% Writing input clauses... done.  (0.62 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 2.3 seconds (prep 0.10 seconds, search 2.20 seconds)

capacity(boat)=2

0:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=3  num(ca,bank2)=0
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=1,howmany(ca)=1)

1:  num(mi,bank1)=2  num(mi,bank2)=1  num(ca,bank1)=2  num(ca,bank2)=1
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=0)

2:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=2  num(ca,bank2)=1
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=2)

3:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=0  num(ca,bank2)=3
```

```
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1)

4:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=1  num(ca,bank2)=2
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0)

5:  num(mi,bank1)=1  num(mi,bank2)=2  num(ca,bank1)=1  num(ca,bank2)=2
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=1)

6:  num(mi,bank1)=2  num(mi,bank2)=1  num(ca,bank1)=2  num(ca,bank2)=1
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0)

7:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=2  num(ca,bank2)=1
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1)

8:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=3  num(ca,bank2)=0
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=2)

9:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=1  num(ca,bank2)=2
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=0)
```

```
10:  num(mi,bank1)=1  num(mi,bank2)=2  num(ca,bank1)=1  num(ca,bank2)=2
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=1,howmany(ca)=1)

11:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=0  num(ca,bank2)=3
loc(boat)=bank2


yes
```

A.2 Solution for Two Boats

```
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 1.05 seconds (prep 0.24 seconds, search 0.81 seconds)

capacity(boat)=2  capacity(boat1)=1

0:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=3  num(ca,bank2)=0
loc(boat)=bank1  loc(boat1)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=2)
cross(boat1,to=bank2,howmany(mi)=0,howmany(ca)=1)  departing(mi,bank1)=0
departing(mi,bank2)=0  departing(ca,bank1)=3  departing(ca,bank2)=0

1:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=0  num(ca,bank2)=3
loc(boat)=bank2  loc(boat1)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1)
cross(boat1,to=bank1,howmany(mi)=0,howmany(ca)=1)  departing(mi,bank1)=0
departing(mi,bank2)=0  departing(ca,bank1)=0  departing(ca,bank2)=2

2:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=2  num(ca,bank2)=1
```

```
loc(boat)=bank1  loc(boat1)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0)
cross(boat1,to=bank2,howmany(mi)=1,howmany(ca)=0)  departing(mi,bank1)=3
departing(mi,bank2)=0  departing(ca,bank1)=0  departing(ca,bank2)=0

3:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=2  num(ca,bank2)=1
loc(boat)=bank2  loc(boat1)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=0)
cross(boat1,to=bank1,howmany(mi)=1,howmany(ca)=0)  departing(mi,bank1)=0
departing(mi,bank2)=2  departing(ca,bank1)=0  departing(ca,bank2)=0

4:  num(mi,bank1)=2  num(mi,bank2)=1  num(ca,bank1)=2  num(ca,bank2)=1
loc(boat)=bank1  loc(boat1)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0)
cross(boat1,to=bank2,howmany(mi)=0,howmany(ca)=1)  departing(mi,bank1)=2
departing(mi,bank2)=0  departing(ca,bank1)=1  departing(ca,bank2)=0

5:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=1  num(ca,bank2)=2
loc(boat)=bank2  loc(boat1)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1)
cross(boat1,to=bank1,howmany(mi)=0,howmany(ca)=1)  departing(mi,bank1)=0
departing(mi,bank2)=0  departing(ca,bank1)=0  departing(ca,bank2)=2

6:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=3  num(ca,bank2)=0
loc(boat)=bank1  loc(boat1)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=2)
cross(boat1,to=bank2,howmany(mi)=0,howmany(ca)=1)  departing(mi,bank1)=0
departing(mi,bank2)=0  departing(ca,bank1)=3  departing(ca,bank2)=0
```

```
7:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=0  num(ca,bank2)=3
loc(boat)=bank2  loc(boat1)=bank2
```

A.3 Solution for Four Missionaries and Four Cannibals

```
% Verifying that the problem is not solvable...


% Verifying the given invariant...

% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 326 atoms (including new atoms), 1186 clauses
% Writing input clauses... done.  (0.08 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0.09 seconds (prep 0.03 seconds, search 0.06 seconds)

No solution with maxstep 1.


% Verified the invariant.


% Verifying that initial state satisfies the invariant...

% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 44 atoms (including new atoms), 91 clauses
% Writing input clauses... done.  (0.04 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0.02 seconds (prep 0.02 seconds, search 0.00 seconds)

capacity(boat)=2
```

```
0:  num(mi,bank1)=4  num(mi,bank2)=0  num(ca,bank1)=4  num(ca,bank2)=0
loc(boat)=bank2



% Initial state satisfies the invariant.


% Verifying that every goal state does not satisfy the invariant...

% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 44 atoms (including new atoms), 91 clauses
% Writing input clauses... done.  (0.04 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0.02 seconds (prep 0.01 seconds, search 0.01 seconds)

No solution with maxstep 0.

% Every goal state does not satisfy the invariant.


% Verified that the problem is not solvable for any number of steps.
```

A.4 Solution for the Boat Carrying Three

A.4.1 Five Pairs

```
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 10.32 seconds (prep 0.25 seconds, search 10.07 seconds)

capacity(boat)=3

0:  num(mi,bank1)=5  num(mi,bank2)=0  num(ca,bank1)=5  num(ca,bank2)=0
```

```
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=1,howmany(ca)=1)

1:  num(mi,bank1)=4  num(mi,bank2)=1  num(ca,bank1)=4  num(ca,bank2)=1
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=0)

2:  num(mi,bank1)=5  num(mi,bank2)=0  num(ca,bank1)=4  num(ca,bank2)=1
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=3)

3:  num(mi,bank1)=5  num(mi,bank2)=0  num(ca,bank1)=1  num(ca,bank2)=4
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1)

4:  num(mi,bank1)=5  num(mi,bank2)=0  num(ca,bank1)=2  num(ca,bank2)=3
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=3,howmany(ca)=0)

5:  num(mi,bank1)=2  num(mi,bank2)=3  num(ca,bank1)=2  num(ca,bank2)=3
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=1)

6:  num(mi,bank1)=3  num(mi,bank2)=2  num(ca,bank1)=3  num(ca,bank2)=2
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=3,howmany(ca)=0)
```

```
7:  num(mi,bank1)=0  num(mi,bank2)=5  num(ca,bank1)=3  num(ca,bank2)=2
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1)

8:  num(mi,bank1)=0  num(mi,bank2)=5  num(ca,bank1)=4  num(ca,bank2)=1
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=3)

9:  num(mi,bank1)=0  num(mi,bank2)=5  num(ca,bank1)=1  num(ca,bank2)=4
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=0)

10:  num(mi,bank1)=1  num(mi,bank2)=4  num(ca,bank1)=1  num(ca,bank2)=4
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=1,howmany(ca)=1)

11:  num(mi,bank1)=0  num(mi,bank2)=5  num(ca,bank1)=0  num(ca,bank2)=5
loc(boat)=bank2
```

A.4.2 Six Pairs

```
% Verifying that the problem is not solvable...


% Verifying the given invariant...

% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 513 atoms (including new atoms), 2013 clauses
% Writing input clauses... done.  (0.20 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
```

```
% Solution time: 0.19 seconds (prep 0.06 seconds, search 0.13 seconds)

No solution with maxstep 1.


% Verified the invariant.


% Verifying that initial state satisfies the invariant...

% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 62 atoms (including new atoms), 146 clauses
% Writing input clauses... done.  (0.13 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0.05 seconds (prep 0.04 seconds, search 0.01 seconds)

capacity(boat)=3

0:  num(mi,bank1)=6  num(mi,bank2)=0  num(ca,bank1)=6  num(ca,bank2)=0
loc(boat)=bank1



% Initial state satisfies the invariant.


% Verifying that every goal state does not satisfy the invariant...

% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 62 atoms (including new atoms), 146 clauses
% Writing input clauses... done.  (0.12 seconds)
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
```

```
% Solution time: 0.04 seconds (prep 0.04 seconds, search 0.00 seconds)

No solution with maxstep 0.

% Every goal state does not satisfy the invariant.


% Verified that the problem is not solvable for any number of steps.
```

A.5 Solution for Converting Cannibals

```
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 9.34 seconds (prep 2.24 seconds, search 7.10 seconds)

capacity(boat)=2

0:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=3  num(ca,bank2)=0
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=2)
departing(mi,bank1)=0  departing(mi,bank2)=0  departing(ca,bank1)=2
departing(ca,bank2)=0

1:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=1  num(ca,bank2)=2
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1)  convert(bank1)
departing(mi,bank1)=0  departing(mi,bank2)=0  departing(ca,bank1)=0
departing(ca,bank2)=1

2:  num(mi,bank1)=4  num(mi,bank2)=0  num(ca,bank1)=1  num(ca,bank2)=1
loc(boat)=bank1
```

```
ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0)
departing(mi,bank1)=2  departing(mi,bank2)=0  departing(ca,bank1)=0
departing(ca,bank2)=0

3:  num(mi,bank1)=2  num(mi,bank2)=2  num(ca,bank1)=1  num(ca,bank2)=1
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=0)
departing(mi,bank1)=0  departing(mi,bank2)=1  departing(ca,bank1)=0
departing(ca,bank2)=0

4:  num(mi,bank1)=3  num(mi,bank2)=1  num(ca,bank1)=1  num(ca,bank2)=1
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0)
departing(mi,bank1)=2  departing(mi,bank2)=0  departing(ca,bank1)=0
departing(ca,bank2)=0

5:  num(mi,bank1)=1  num(mi,bank2)=3  num(ca,bank1)=1  num(ca,bank2)=1
loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=0)
departing(mi,bank1)=0  departing(mi,bank2)=1  departing(ca,bank1)=0
departing(ca,bank2)=0

6:  num(mi,bank1)=2  num(mi,bank2)=2  num(ca,bank1)=1  num(ca,bank2)=1
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=1,howmany(ca)=1)
departing(mi,bank1)=1  departing(mi,bank2)=0  departing(ca,bank1)=1
departing(ca,bank2)=0

7:  num(mi,bank1)=1  num(mi,bank2)=3  num(ca,bank1)=0  num(ca,bank2)=2
loc(boat)=bank2
```

```
ACTIONS:  cross(boat,to=bank1,howmany(mi)=1,howmany(ca)=0)
departing(mi,bank1)=0  departing(mi,bank2)=1  departing(ca,bank1)=0
departing(ca,bank2)=0

8:  num(mi,bank1)=2  num(mi,bank2)=2  num(ca,bank1)=0  num(ca,bank2)=2
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0)
departing(mi,bank1)=2  departing(mi,bank2)=0  departing(ca,bank1)=0
departing(ca,bank2)=0

9:  num(mi,bank1)=0  num(mi,bank2)=4  num(ca,bank1)=0  num(ca,bank2)=2
loc(boat)=bank2
```

A.6 Solution for Walking on Water

```
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 1.59 seconds (prep 0.31 seconds, search 1.28 seconds)

capacity(boat)=2

0:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=3  num(ca,bank2)=0
num(jc,bank1)=1  num(jc,bank2)=0  loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=2,howmany(jc)=0)
departing(mi,bank1)=0  departing(mi,bank2)=0  departing(ca,bank1)=2
departing(ca,bank2)=0  departing(jc,bank1)=0  departing(jc,bank2)=0

1:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=1  num(ca,bank2)=2
num(jc,bank1)=1  num(jc,bank2)=0  loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1,howmany(jc)=0)
```

```
departing(mi,bank1)=0  departing(mi,bank2)=0  departing(ca,bank1)=0
departing(ca,bank2)=1  departing(jc,bank1)=0  departing(jc,bank2)=0

2:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=2  num(ca,bank2)=1
num(jc,bank1)=1  num(jc,bank2)=0  loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=2,howmany(jc)=0)
departing(mi,bank1)=0  departing(mi,bank2)=0  departing(ca,bank1)=2
departing(ca,bank2)=0  departing(jc,bank1)=0  departing(jc,bank2)=0

3:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=0  num(ca,bank2)=3
num(jc,bank1)=1  num(jc,bank2)=0  loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1,howmany(jc)=0)
departing(mi,bank1)=0  departing(mi,bank2)=0  departing(ca,bank1)=0
departing(ca,bank2)=1  departing(jc,bank1)=0  departing(jc,bank2)=0

4:  num(mi,bank1)=3  num(mi,bank2)=0  num(ca,bank1)=1  num(ca,bank2)=2
num(jc,bank1)=1  num(jc,bank2)=0  loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0,howmany(jc)=0)
departing(mi,bank1)=2  departing(mi,bank2)=0  departing(ca,bank1)=0
departing(ca,bank2)=0  departing(jc,bank1)=0  departing(jc,bank2)=0

5:  num(mi,bank1)=1  num(mi,bank2)=2  num(ca,bank1)=1  num(ca,bank2)=2
num(jc,bank1)=1  num(jc,bank2)=0  loc(boat)=bank2

ACTIONS:  cross(boat,to=bank1,howmany(mi)=0,howmany(ca)=1,howmany(jc)=0)
walk(walk_to=bank2)  departing(mi,bank1)=1  departing(mi,bank2)=0
departing(ca,bank1)=0  departing(ca,bank2)=1  departing(jc,bank1)=1
departing(jc,bank2)=0

6:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=2  num(ca,bank2)=1
num(jc,bank1)=0  num(jc,bank2)=1  loc(boat)=bank1
```

```
ACTIONS:  cross(boat,to=bank2,howmany(mi)=0,howmany(ca)=2,howmany(jc)=0)
departing(mi,bank1)=0  departing(mi,bank2)=0  departing(ca,bank1)=2
departing(ca,bank2)=0  departing(jc,bank1)=0  departing(jc,bank2)=0

7:  num(mi,bank1)=0  num(mi,bank2)=3  num(ca,bank1)=0  num(ca,bank2)=3
num(jc,bank1)=0  num(jc,bank2)=1  loc(boat)=bank2
```

A.7 Solution for the Bridge

```
% Calling mChaff spelt3... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0.7 seconds (prep 0.29 seconds, search 0.41 seconds)

capacity(boat)=2

0:  num(mi,bank1)=5  num(mi,bank2)=0  num(ca,bank1)=5  num(ca,bank2)=0
loc(boat)=bank1

ACTIONS:  useBridge(useBridge_from=bank1,useBridge_to=bank2,
useBridge_howmany(mi)=0,useBridge_howmany(ca)=2)  departing(mi,bank1)=0
departing(mi,bank2)=0  departing(ca,bank1)=2  departing(ca,bank2)=0

1:  num(mi,bank1)=5  num(mi,bank2)=0  num(ca,bank1)=3  num(ca,bank2)=2
loc(boat)=bank1

ACTIONS:  useBridge(useBridge_from=bank1,useBridge_to=bank2,
useBridge_howmany(mi)=2,useBridge_howmany(ca)=0)  departing(mi,bank1)=2
departing(mi,bank2)=0  departing(ca,bank1)=0  departing(ca,bank2)=0

2:  num(mi,bank1)=3  num(mi,bank2)=2  num(ca,bank1)=3  num(ca,bank2)=2
loc(boat)=bank1

ACTIONS:  cross(boat,to=bank2,howmany(mi)=2,howmany(ca)=0)
```

useBridge(useBridge_from=bank1,useBridge_to=bank2,useBridge_howmany(mi)=1,
useBridge_howmany(ca)=1) departing(mi,bank1)=3 departing(mi,bank2)=0
departing(ca,bank1)=1 departing(ca,bank2)=0

3: num(mi,bank1)=0 num(mi,bank2)=5 num(ca,bank1)=2 num(ca,bank2)=3
loc(boat)=bank2

ACTIONS: useBridge(useBridge_from=bank1,useBridge_to=bank2,
useBridge_howmany(mi)=0,useBridge_howmany(ca)=2) departing(mi,bank1)=0
departing(mi,bank2)=0 departing(ca,bank1)=2 departing(ca,bank2)=0

4: num(mi,bank1)=0 num(mi,bank2)=5 num(ca,bank1)=0 num(ca,bank2)=5
loc(boat)=bank2

Vita

Joohyung Lee was born in TaeGu, Korea in 1972, the first son of Soo-Ung Lee and Chun-Hee Kim. In 1992 he entered Seoul National University, where he received a B.S. degree in computer engineering. In 1998 he came to the United States for graduate studies in computer science at the University of Texas at Austin. He was married to Jee Hyun Park in 2002, and they expect the first baby in June 2005.

He has published the following papers:

1. Joohyung Lee and Vladimir Lifschitz. Describing Additive Fluents in Action Language $\mathcal{C}+$. In *Proc. Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1079–1084, 2003.

2. Joohyung Lee and Vladimir Lifschitz. Loop Formulas for Disjunctive Logic Programs. In *Proc. Nineteenth International Conference on Logic Programming (ICLP-03)*, pages 451–465, 2003.

3. Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain and Hudson Turner. Nonmonotonic Causal Theories. *Artificial Intelligence*, 153:49–104, 2004.

4. Varol Akman, Selim T. Erdoğan, Joohyung Lee, Vladimir Lifschitz and Hud-

son Turner. Representing the Zoo World and the Traffic World in the Language of the Causal Calculator. *Artificial Intelligence*, 153:105–140, 2004.

5. Joohyung Lee. Nondefinite vs. Definite Causal Theories. In *Proc. Seventh International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-04)*, pages 141–153, 2004.

6. Joohyung Lee and Fangzhen Lin. Loop Formulas for Circumscription. In *Proc. Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 281–286, 2004.

7. Joohyung Lee. A Model-Theoretic Counterpart of Loop Formulas. In *Proc. Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005, to appear.

Permanent Address: BoonDang-Gu JungJa-Dong WooSung #406-1101
SungNam-Si, KyungKi-Do, South Korea, 463–752

This dissertation was typeset with LaTeX 2_ε by the author.

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.

[Apt and Bezem, 1990] Krzysztof Apt and Marc Bezem. Acyclic programs. In David Warren and Peter Szeredi, editors, *Proceedings of International Conference on Logic Programming (ICLP)*, pages 617–633, 1990.

[Armando and Compagna, 2002] Alessandro Armando and Luca Compagna. Automatic sat-compilation of protocol insecurity via reduction to planning. In *Proc. Joint Int'l Conference on Formal Techniques for Networked and Distributed Systems 2002*, 2002.

[Artikis *et al.*, 2003a] A. Artikis, M. Sergot, and J. Pitt. An executable specification of an argumentation protocol. In *Proceedings of Conference on Artificial Intelligence and Law (ICAIL)*, pages 1–11. ACM Press, 2003.

[Artikis *et al.*, 2003b] A. Artikis, M. Sergot, and J. Pitt. Specifying electronic societies with the Causal Calculator. In F. Giunchiglia, J. Odell, and G. Weiss,

editors, *Proceedings of Workshop on Agent-Oriented Software Engineering III (AOSE)*, LNCS 2585. Springer, 2003.

[Baker, 1991] Andrew Baker. Nonmonotonic reasoning in the framework of situation calculus. *Artificial Intelligence*, 49:5–23, 1991.

[Baral and Gelfond, 1997] Chitta Baral and Michael Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31, 1997.

[Bayardo and Schrag, 1997] Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. IJCAI-97*, pages 203–208, 1997.

[Campbell and Lifschitz, 2003] Jonathan Campbell and Vladimir Lifschitz. Reinforcing a claim in commonsense reasoning.[1] In *Working Notes of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 2003.

[Chopra and Singh, 2003] Amit Chopra and Munindar Singh. Nonmonotonic commitment machines. In *Agent Communication Languages and Conversation Policies AAMAS 2003 Workshop*, 2003.

[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.

[Cook, 1971] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. Third Annual ACM Symposium on Theory of Computing*, 1971.

[Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A ma-

[1] `http://www.cs.utexas.edu/users/vl/papers/sams.ps` .

chine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Davis, 1990] Ernest Davis. *Representations of Commonsense Knowledge*. Morgan Kaufmann, 1990.

[Doğandağ *et al.*, 2004] Semra Doğandağ, Paolo Ferraris, and Vladimir Lifschitz. Almost definite causal theories.[2] In *Proc. 7th Int'l Conference on Logic Programming and Nonmonotonic Reasoning*, pages 74–86, 2004.

[Erdem and Lifschitz, 2003] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[Eshghi and Kowalski, 1989] Kave Eshghi and Robert Kowalski. Abduction compared with negation as failure. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of International Conference on Logic Programming (ICLP)*, pages 234–255, 1989.

[Evans, 1989] Chris Evans. Negation-as-failure as an approach to the Hanks and McDermott problem. In *Proc. Second Int'l Symp. on Artificial Intelligence*, 1989.

[Fages, 1994] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[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.

[2] `http://www.cs.utexas.edu/users/vl/papers/adct.ps` .

[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 *Proc. AAAI-90*, pages 524–530. AAAI Press, 1990.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080, 1988.

[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.[3] *Electronic Transactions on AI*, 3:195–210, 1998.

[Gelfond, 1987] Michael Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[Gelfond, 1989] Michael Gelfond. Autoepistemic logic and formalization of commonsense reasoning. In Michael Reinfrank, Johan de Kleer, Matthew Ginsberg, and Erik Sandewall, editors, *Non-Monotonic Reasoning: 2nd Int'l Workshop (Lecture Notes in Artificial Intelligence 346)*, pages 176–186. Springer-Verlag, 1989.

[Giunchiglia and Lifschitz, 1998] Enrico Giunchiglia and Vladimir Lifschitz. An ac-

[3] `http://www.ep.liu.se/ea/cis/1998/016/` .

tion language based on causal explanation: Preliminary report. In *Proc. AAAI-98*, 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.

[Hanks and McDermott, 1987] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[Hayes, 1977] Patrick Hayes. In defence of logic. In *Proc. IJCAI-77*, 1977.

[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.

[Kautz and Selman, 1996] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic and stochastic search. In *Proc. AAAI-96*, pages 1194–1201, 1996.

[Kautz and Walser, 1999] Henry Kautz and Joachim Walser. State-space planning by integer optimization. In *Proc. AAAI-99*, pages 526–533, 1999.

[Koehler, 1998] Jana Koehler. Planning under resource constraints. In *Proc. ECAI-98*, pages 489–493, 1998.

[Lee and Lifschitz, 2003] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. ICLP-03*, pages 451–465, 2003.

[Lee, 2004] Joohyung Lee. Nondefinite vs. definite causal theories. In *Proc. 7th Int'l Conference on Logic Programming and Nonmonotonic Reasoning*, pages 141–153, 2004.

[Lee, 2005] Joohyung Lee. A model-theoretic counterpart of loop formulas. In *Proc. IJCAI*, 2005. To appear.

[Lifschitz and Razborov, 2004] Vladimir Lifschitz and Alexander Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 2004. To appear.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proceedings of International Conference on Logic Programming (ICLP)*, pages 23–37, 1994.

[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, 1987] Vladimir Lifschitz. Formal theories of action (preliminary report). In *Proc. IJCAI-87*, pages 966–972, 1987.

[Lifschitz, 1991] Vladimir Lifschitz. Towards a metatheory of action. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proc. Second Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 376–386, 1991.

[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 *Principles of Knowledge Representation and Reasoning: Proc. Seventh Int'l Conf.*, pages 85–96, 2000.

[Lin and Zhao, 2004] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157:115–137, 2004.

[Lin, 1995] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proc. IJCAI-95*, 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 *Proc. AAAI-97*, pages 460–465, 1997.

[McCain and Turner, 1998] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proc. Sixth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 212–223, 1998.

[McCain, 1997] Norman McCain. *Causality in Commonsense Reasoning about Actions.*[4] PhD thesis, University of Texas at Austin, 1997.

[4] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.Z` .

[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 *Proc. Teddington Conf. on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationery Office.

[McCarthy, 1980] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39,171–172, 1980. Reproduced in [McCarthy, 1990].

[McCarthy, 1986] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986. Reproduced in [McCarthy, 1990].

[McCarthy, 1990] John McCarthy. *Formalizing Common Sense: Papers by John McCarthy.* Ablex, Norwood, NJ, 1990.

[McCarthy, 1998] John McCarthy. Elaboration tolerance.[5] In progress, 1998.

[McDermott and Doyle, 1980] Drew McDermott and Jon Doyle. Nonmonotonic logic I. *Artificial Intelligence*, 13:41–72, 1980.

[Moore, 1985] Robert Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[5] `http://www-formal.stanford.edu/jmc/elaboration.html` .

[Morris, 1988] Paul Morris. The anomalous extension problem in default reasoning. *Artificial Intelligence*, 35(3):383–399, 1988.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC-01*, 2001.

[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.

[Pearl, 1988] Judea Pearl. *Causality.* Cambridge University Press, 1988.

[Pednault, 1994] Edwin Pednault. ADL and the state-transition model of action. *Journal of Logic and Computation*, 4:467–512, 1994.

[Przymusinski, 1989] Teodor Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Reiter, 1978] Raymond Reiter. On closed world data bases. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum Press, New York, 1978.

[Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Reiter, 1991] Raymond Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[Turner, 1996] Hudson Turner. Splitting a default theory. In *Proc. AAAI-96*, pages 645–651, 1996.

[Turner, 1997] Hudson Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

[Turner, 1999] Hudson Turner. A logic of universal causation. *Artificial Intelligence*, 113:87–123, 1999.

[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.

[Wolfman and Weld, 1999] Steven Wolfman and Daniel Weld. The LPSAT engine and its application to resource planning. In *Proc. IJCAI-99*, pages 310–316, 1999.

[Zhang *et al.*, 2001] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proc. ICCAD-01*, pages 279–285, 2001.

[Zhang, 1997] Hantao Zhang. An efficient propositional prover. In *Proc. CADE-97*, 1997.