The Dissertation Committee for Fangkai Yang
certifies that this is the approved version of the following dissertation:

Representing Actions in Logic-Based Languages

Committee:

Vladimir Lifschitz, Supervisor

Chitta Baral

E. Allen Emerson

Bruce Porter

Peter Stone

Representing Actions in Logic-Based Languages

by

Fangkai Yang, B.E.; M.E.

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 2014

To my family.

Acknowledgments

This dissertation would not have been possible without the continuous support and encouragement of Vladimir Lifschitz. During the many years that I have worked with Vladimir, he has been a wise advisor, an effective colleague, and a genuine friend. He has exemplary taste when it comes to scientific research, and I always strive to emulate it: stick to important problems with maximal scientific value, dedicate yourself to and enjoy the process of scientific exploration, pay attention to detail, and appreciate mathematical rigor, simplicity and elegance. I feel very privileged to have had the opportunity to be advised by him; it has been an invaluable experience and deeply influential to my career. Thanks also go to Elena Lifschitz for her care and motherly love!

I am also thankful to the other members of my dissertation committee: Chitta Baral, E. Allen Emerson, Bruce Porter, and Peter Stone for careful reading of this dissertation and useful comments.

In finishing the last part of this dissertation, I have benefited greatly from discussions with Michael Gelfond, Peter Stone and Torsten Schaub. Throughout the years, as I tried to apply logic programming to autonomous agents, Michael kindly and patiently answered all of my questions regarding his previous work. He also helped me to understand the ASP encodng of the

RCS system when I tried to reformulate it in $\mathfrak{BC}$. Peter has enthusiastically encouraged me to explore the possibility of applying action theory to mobile robots in the Building-wide Intelligence project, and introduced me to the exciting world of learning robots. His insightful advice about engineering real robots has broadened my views and drawn my attention from the theoretical aspects of logic-based artificial intelligence to practical challenges. Torsten kindly and patiently answered my numerous questions about tuning answer set solvers.

It has been an enormous fortune for me to be able to work with a group of incredibly energetic and intelligent colleagues. They include the members of Texas Action Group: Esra Erdem, Selim Erdoğan, Paolo Ferraris, Amelia Harrison, Joohyung Lee, and Yuliya Lierler. I particularly appreciate the friendship I've shared with Yuliya Lierler, with whom I am always happy to share my opinions about science, life and career, and from whom I get invaluable advice. Thanks also go to the members of the Learning Agent Research Group who are involved in the Building-Wide Intelligence project: Piyush Khandelwal, Matteo Leonetti, and Shiqi Zhang.

Over the years I have benefited from many delightful scientific and personal communications with my colleagues worldwide: Joseph Babb, Marcello Balduccini, Xiaoping Chen, Martin Gebser, Gregory Gelfond, Guoqiang Jin, Yunsong Meng, Roland Kaminski,Volkan Patoglu, Tran Cao Son, Yan Zhang, Yi Zhou. I am grateful to all of them.

I would like to thank the Department of Computer Science and Artificitial Intelligence Lab for providing me with an excellent facility in which to conduct research, the world-class faculty members with whom I took classes which broadened my knowledge about many advanced topics of computer science, and the friendly staff. I am particularly thankful to Warren Hunt, for generously donating the workstation with which I finish this dissertation.

I am greatly indebted to my family members and my deepest thanks are undoubtedly reserved for them. I would like to thank my parents Qi Fang and Liangfeng Zhang for their endless love and support of my life, study and career plan. They are always by my side whenever I need help or advice, sharing my success and frustrations, and encouraging me to define and explore my path. I am equally thankful to my brother Jianbu Zhang and sister Hongyan Zhang, my sister-in-law Jing Geng and my brother-in-law Shengli Wang. During the past years as I worked towards my degree in a country 10,000 miles away from my home, it was not possible for me to visit my parents as much as I should have. They took great care of my parents, bringing them endless happiness and warmth, compensating for the fact that I could not be always around.

Fangkai Yang

The University of Texas at Austin

May 2014

Representing Actions in Logic-Based Languages

Publication No. ____________

Fangkai Yang, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Vladimir Lifschitz

Knowledge about actions is an important part of commonsense knowledge studied in Artificial Intelligence. For decades, researchers have been developing methods for describing how actions affect states of the world and for automating reasoning about actions. In recent years, significant progress has been made. In particular, the frame problem has been solved using nonmonotonic knowledge representation formalisms, such as logic programming under the answer set semantics. New theories of causality have allowed us to express causal dependencies between fluents, which has proved essential for solving the ramification problem. It has been shown that reasoning about actions described by logic programs and causal theories can be automated using answer set programming.

Action description languages are high level languages that allow us to represent knowledge about actions more concisely than when logic programs are used. Many action description languages have been described in the literature, including $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{C}+$. Reasoning about dynamic domains described in languages $\mathcal{C}$ and $\mathcal{C}+$ can be performed automatically using the Causal Calculator (CCALC), which employs SAT solvers for search, and the systems COALA and CPLUS2ASP, which employ answer set solvers such as CLINGO.

The dissertation addresses problems of three kinds. First, we study some mathematical properties of expressive action languages based on nonmonotonic causal logic that were not well understood until now. This includes causal rules expressing synonymy, nondefinite causal rules, and nonpropositional causal rules. We generalize existing translations from nonmonotonic causal theories to logic programming under the answer set semantics. This makes it possible to automate reasoning with a wider class of causal theories by calling answer set solvers.

Second, we design and study a new action language $\mathcal{BC}$, which is more expressive in some ways than the existing and previously proposed languages. We develop a framework that combines the most useful expressive features of the languages $\mathcal{B}$ and $\mathcal{C}+$, and use program completion to characterize the effects of actions described in these languages.

Third, we illustrate the possibilities of the new action language by two practical applications: to the dynamic domain of the Reactive Control System of the space shuttle, and to the task planning of mobile robots.

Table of Contents

List of Figures

Chapter 1

Introduction

Knowledge about actions is an important part of commonsense knowledge studied in Artificial Intelligence. For decades, researchers have been developing methods for describing how actions affect states of the world and for the automation of reasoning about actions. Following the groundbreaking paper by John McCarthy and Patrick Hayes [62], the logic school of artificial intelligence describes a state of the world by values of *fluents* that can be affected by performing actions. For instance, a light switch can be either on or off, so that its state can be described by the Boolean (that is, truth-valued) fluent On. The location of a block B in the blocks world can be described by the non-Boolean fluent $Loc(B)$. Possible values of this fluent are $Table$ and other blocks. Performing the action $Toggle$ changes the value of On from *true* to *false* and vice versa, and performing the action $Move(B, Table)$ makes the value of $Loc(B)$ equal to $Table$.

Reasoning about dynamic domains typically deals with three kinds of computational problems:

- temporal projection (also known as prediction): given the values of fluents in the initial state and a sequence of actions to be performed, what can we say about the values of fluents in the resulting state?

- postdiction: given the values of fluents after performing a sequence of actions, what can we say about the initial state?

- planning: find a sequence of actions that leads from a given initial state to a given goal state.

An agent should be able to answer queries of these kinds in order to behave intelligently in the world.

Developing methodologies for describing action domains in logic-based formalisms and for solving computational problems described above are major topics of research in artificial intelligence. After the publication of the paper by McCarthy and Hayes [62], solving the frame problem remained one of the key research challenges, leading to the invention of default theories [68], autoepistemic logic [64], and circumscription [60, 61]. One of the solutions to the frame problem available today uses the answer set semantics of logic programs [27, 28]. Another approach to the frame problem is based on theories of causality [25, 33, 44, 54, 59].

Action description languages are high level languages that allow us to represent knowledge about actions more concisely than the situation calculus. Many action description languages have been described in the literature, from the well-known STRIPS [22], to more expressive ADL [66], to action languages invented more recently such as $\mathcal{B}$ [30, Section 5], $\mathcal{C}$ [30, 34, Section 6], and

$\mathcal{C}$+ [33]. The last three allow us to describe actions with indirect effects, or "ramifications." The semantics of $\mathcal{B}$ is closely related to the answer set semantics of logic programs. The languages $\mathcal{C}$ and $\mathcal{C}$+ are closely related to the causal logic proposed in [59].

Theoretical work on developing logic-based formalisms and action languages has led to the implementation of systems that can efficiently automate reasoning about actions. The "definite" fragment of $\mathcal{C}$+ is implemented in the Causal Calculator (CCALC)[1], which translates a causal theory into a set of propositional clauses and calls a SAT solver to answer queries. On the other hand, definite causal theories can be represented by logic program under the answer set semantics [58], so that they can be also implemented using computational methods of answer set programming (ASP). Systems of this kind are COALA [24] and CPLUS2ASP [2, 8]. They transform an action description and a query into a logic program and call an answer set solver, such as CLINGO with its gounder GRINGO[2], or SMODELS with its grounder LPARSE[3] to compute its stable models.

Some kinds of commonsense reasoning require the use of nondefinite causal theories. It was pointed out in [14] that humans often define actions as special cases of a more general action. For instance, the dictionary defines action "push" as synonymous with "move by steady pressure." To talk about

[1]www.cs.utexas.edu/user/tag/cc/
[2]http://potassco.sourceforge.net/
[3]http://www.tcs.hut.fi/Software/smodels/

synonymy in causal logic, we need nondefinite rules.

Both stable models and causal theories were originally defined in propositional setting. These concepts were lifted to the first-order case in [44] and [17, 20]. The first-order versions of these nonmonotonic formalisms are similar to circumscription: each of them employs a syntactic transformation that turns a first-order sentence into a stronger sentence, which may involve second-order quantifiers. This additional generality is important for many applications to the theory of commonsense reasoning.

The dissertation addresses problems of three kinds. First, we study some mathematical properties of expressive action languages based on nonmonotonic causal logic that were not well understood until now. This includes causal rules expressing synonymy, nondefinite causal rules, and nonpropositional causal rules. In particular, we generalize existing translations from nonmonotonic causal theories to logic programming under the answer set semantics, so that it is possible to automate reasoning with a wider class of causal theories by calling answer set solvers.

Second, we design and study a new action language $\mathcal{BC}$, more expressive in some ways than the languages proposed in the past. This includes developing a framework that combines the most useful expressive possibilities of the languages $\mathcal{B}$ and $\mathcal{C}$, and using program completion to characterize the effects of actions described in these languages.

Third, we illustrate the possibilities of the new action language by two practical applications: to the dynamic domain of the Reactive Control System of the space shuttle, and to the task planning of mobile robots. The system CPLUS2ASP is used for translating action descriptions into logic programs for query-answering and automated planning.

The dissertation is organized as follows. Chapter 2 reviews earlier work on reasoning about actions, in particular, nonmonotonic formalisms, and the technical background for the research described in this dissertation: first-order nonmonotonic causal logic [44] and first-order stable models [20].

Chapters 3–5 focuses on representing actions using causal logic. Chapter 3 deals with the important concept of literal completion [44]. As pointed out above, the semantics of causal theories involves second-order formulas. Literal completion is the process that allows us, in some cases, to turn them into equivalent first-order formulas. It is, however, applicable to a causal theory only if each of its "explainable" symbols is a predicate constant. Explainable function symbols are often useful: they are needed, for instance, to talk about non-Boolean fluents such as $Loc(B)$ in the blocks world example above. We show in this chapter how to extend the definition of literal completion and the theorem on literal completion from [44] to causal theories with explainable function symbols.

In Chapter 4, we generalize earlier work on translating causal theories into logic programs. First, we study causal rules with heads of the form $L_1 \leftrightarrow L_2$, where L_1 and L_2 are literals. Such a rule says that there is a cause for L_1 and L_2 to be equivalent ("synonymous") under some condition, expressed by the body of the rule. As described above, synonymy rules play an important role in the theory of commonsense reasoning in view of the fact that humans often explain the meaning of words by referring to their synonyms. Second, we extend the translations proposed earlier from propositional causal rules to first-order causal rules.

In Chapter 5, we propose a way to eliminate explainable function symbols in causal theories in favor of explainable predicate symbols. This is important because the translation proposed in Chapter 4 is not directly applicable to fluents with non-Boolean values, represented by function symbols. In classical logic, this process is well understood, but extending it to nonmonotonic causal logic is not straightforward, especially if we want to arrive eventually at an executable ASP program. In this chapter, we describe two procedures for eliminating function constants from a causal theory in favor of predicate constants, "general" and "definite." Then we show how definite elimination can help us turn a causal theory into executable ASP code, and how it can be extended to rules that express the synonymy of function symbols.

Chapters 6–9 focus on representing actions by logic programs. In Chapter 6, we define a new action description language, called $\mathcal{BC}$, that combines the attractive features of $\mathcal{B}$ and $\mathcal{C}+$. This language, like $\mathcal{B}$, can be implemented using computational methods of answer set programming. The main difference between the way causal rules are treated in $\mathcal{B}$ and in $\mathcal{BC}$ is similar to the difference between inference rules and default rules. Informally speaking, a default rule allows us to derive its conclusion from its premise if its "justifications" can be consistently assumed; default logic [68] makes this idea precise.

The semantics of $\mathcal{BC}$ is defined by transforming action descriptions into logic programs under the stable model semantics. We define two translations from $\mathcal{BC}$ into logic programming. Their target languages use slightly different versions of the stable model semantics, but we show that they give the same meaning to $\mathcal{BC}$-descriptions. The first version uses nested occurrences of negation as failure [49]; the second involves strong (classical) negation [28] but does not require nesting.

Examples of formalizing commonsense domains discussed in this chapter illustrate the expressive capabilities of $\mathcal{BC}$ and the use of answer set solvers for the automation of reasoning about actions described in this language. We study properties of language $\mathcal{BC}$, including two theorems relating $\mathcal{BC}$ to $\mathcal{B}$ and to $\mathcal{C}+$.

Proofs of the theorems stated in Chapter 6 are presented in Chapter 7.

Chapter 8 is about conditions when stable models of a logic program are characterized by the program's completion in the sense of [11, 56]. This problem was discussed in many papers by many researchers, beginning with François Fages [15]. Our enhancement of Fages' theorem, based on the concept of a "rule dependency graph," is motivated by examples of logic programs describing dynamic domains. As a tool for proving Theorem 8.2.1 of that chapter, in Chapter 9 we extend Fages' theorem to infinitary propositional logic programs. Sections 9.1–9.3, in which we review Truszczynski's definition of a stable model for infinitary formulas [72] and prove a Fages-style theorem for infinitary programs, can be read independently of the rest of this dissertation.

Chapters 10–12 describe practical applications of action language $\mathcal{BC}$. In these applications, dynamic domains are formalized as action descriptions, and CPLUS2ASP is used to translate action descriptions into logic programming, so that automated planning and query-answering can be achieved by calling an answer set solver. CPLUS2ASP was first implemented as a tool for translating CCALC input into the input language of GRINGO [8]. Its second version also supports an input language with rules similar to the laws in $\mathcal{BC}$. In Chapter 10, we describe a subset of the input language of CPLUS2ASP and relate it to $\mathcal{BC}$ action descriptions. This input language extends $\mathcal{BC}$ by several useful syntactical features, leading to more concise formalizations.

In Chapter 11 we formalize the Reactive Control System (RCS) of space shuttle. The primary responsibility of the RCS system is for maneuvering the space shuttle when it is in space. It consists of fuel and oxidizer tanks, valves and other plumbing needed to provide propellant to the maneuvering jets of the shuttle. It also consists of electronic circuitry: both to control valves in the fuel lines and to prepare the jets to receive firing commands. Both human and computer can perform a sequence of actions either by issuing commands or manual flipping the switches to control the valves so that fuel and oxidizer is pumped to jets to perform certain maneuver mission. USA/RCS-Advisor [65] is a system such that the sequence of actions can be automatically generated. In that system, the dynamic domain is formalized in logic programming. Our formalization is based on the action language described in Chapter 10. By comparing the CPLUS2ASP encoding and the original ASP encoding, we illustrate the expressiveness of $\mathcal{BC}$.

Chapter 12 describes the use of $\mathcal{BC}$ to implementing a task planner for a mobile robot that behaves intelligently in the newly built Dell and Gates Computer Science Complex (GDC) of the University of Texas at Austin. The project, named Building-Wide Intelligence and led by Professor Peter Stone, aims to build a team of mobile robots that will help both the long-term inhabitants and visitors to the building. The task planner allows the robot to plan for several tasks, including collecting mails from people, asking about locations of people in case of incomplete knowledge by human-robot interaction, and learning action costs through experience. Execution monitoring and replanning are implemented to handle execution failure. The task planner implemented based on the approach is used on a robot operating in the 3NE Wing of the GDC building.

Most work described in the dissertation is published [19, 37, 39, 40, 50–53, 73].

Chapter 2

Background

2.1 Earlier Work on Reasoning about Actions

The *Situation Calculus* [62] is a formalism for describing fluents and actions based on first-order logic. As soon as it was proposed, researchers noticed that it was not easy to use it because of the need to specify which conditions are *not* affected by executing actions. In human common sense, the general rule-of-thumb is:

> *an action can be assumed not to change a given property of a situation unless there is evidence to the contrary.*[1]

This is known as the *commonsense law of inertia*. It will allow us to derive, for instance, that throwing a ball will not change its color. Reasoning with the commonsense law of inertia is *not monotonic*: with more information added later, some conclusions derived earlier can be retracted, because that information may provide "evidence to the contrary." For instance, the color of the ball will change if it is thrown into a bucket of paint. However, first-order logic is known to be monotonic, so that formalizing the commonsense law of inertia requires a declarative language of a different kind.

[1] http://plato.stanford.edu/entries/frame-problem/

John McCarthy [60, 61] proposed an approach to the frame problem based on "abnormality theories"—first-order theories containing the special predicate Ab. A model of an abnormality theory is called minimal if the extent of Ab in it cannot be made smaller without violating the axioms.[2] Minimal models are models "with few abnormal objects." The minimality condition can be expressed by a second-order formula, called *circumscription.* McCarthy proposed a formalization of the commonsense law of inertia based on circumscription, which expressed the idea of minimizing change. But his approach turned out to be incorrect: the theory has unintended minimal models that do not have the desired properties. The fact that some unintuitive models are not eliminated by minimizing change was proved using a counter example known as the Yale Shooting Scenario [36].

Reiter's default theories [68] can postulate both first-order theory axioms and *defaults* — expressions of the form

$$\frac{F:\ \mathsf{M}\,G_1\ \cdots\ \mathsf{M}\,G_n}{H},$$

where the *premises* F, the *justifications* $G_1, \ldots, G_n$ and the *conclusion* H are first-order formulas. Intuitively this default allows us to derive H from F if all justifications can be consistently assumed. Reiter expressed the commonsense law of inertia by a "frame default."

[2] "Smaller" is understood here in the sense of set inclusion, not in the sense of comparing cardinalities. To make the concept of a minimal model precise, we need to specify whether the extents of predicates other than Ab may be changed as we try to make the extent of Ab smaller; we do not go here into discussing this issue.

Prolog rules with negation as failure are similar to Reiter's defaults [6]. The concept of an answer set [27, 28] provided a precise semantics for negation as failure in Prolog rules and established an even closer relationship between Prolog rules and Reiter's default theories. Logic programming under the answer set semantics, known as *answer set programming*, was later used to describe dynamic domains and reasoning about actions [29].

Theories of causality by Geffner [25] and Lin [54] allow us to express causal dependencies between fluents. Nonmonotonic causal theories in the sense of McCain and Turner [59] and their generalizations [33, 44] achieve this goal also. They consist of *causal laws* of the form

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

where F and G are formulas of the underlying signature. The semantics of causal logic distinguishes between "being true" and "being caused". Rule (2.1) is read as "If G is true, then there is a cause for F to be true". The models of a causal theory are interpretations such that anything that is true is caused, and vice versa. This is called the *universal law of causation.* This semantics, like default logic and the answer set semantics, provide a concise solution to the frame problem.

Other solutions to the frame problem were described in [71].

2.2 Examples of Dynamic Domains

Example 2.2.1 (Toggling Switches)**.** There are n light switches. Each of them can be on or off, so the system has 2^n states. The action of toggling a switch changes its position from on to off and vice versa. Any combination of actions can be concurrently executed.

Example 2.2.2 (Moving Objects)**.** There are m objects and each of them can be in one of n possible locations. The system has m^n states. The action of moving an object to a location changes its location, and any combination of actions can be concurrently executed.

Example 2.2.3 (Blocks World)**.** Consider n blocks. In any state, blocks are arranged in towers resting on the table. For instance, if $n = 2$ then there are 3 possible states: Block 1 on top of Block 2; Block 2 on top of Block 1; both blocks are on the table. The action of moving a block changes the location of the block. A block with no block on its top can be moved either to table or to the top of another block whose top is also clear. In this domain, it is possible that some actions can be executed one-by-one but not concurrently. For instance, it is not possible to move two blocks to the same location.

Example 2.2.4 (Leaking Container)**.** Consider a container of capacity n that has a leak, so that it loses k units of liquid per unit of time, unless more liquid is added. With the discrete model of time, the system has $n + 1$ states that describe the amount of liquid inside the container, from 0 to n. The action of filling the container makes the container full: the amount of liquid is turned to n. This domain does not satisfy the commonsense law of inertia: its state changes even when no actions are executed.

Work on representing dynamic domains has been applied also to much larger, realistic domains, and found important practical applications, for instance, the decision support system for the space shuttle discussed in the introduction.

2.3 Review of Causal Logic

2.3.1 Syntax and Semantics

According to [44], a first-order causal theory T is defined by

- a list $\mathbf{c}$ of distinct function and/or predicate constants,[3] called the *explainable symbols* of T, and
- a finite set of *causal rules* of the form $F \Leftarrow G$, where F and G are first-order formulas.

The semantics of causal theories is defined by a syntactic transformation that is somewhat similar to circumscription [61]; its result is usually a second-order formula. For each member c of $\mathbf{c}$, choose a new variable vc similar to c,[4]

[3]We view object constants as function constants of arity 0, so that they are allowed in $\mathbf{c}$. Similarly, propositional symbols are viewed as predicate constants of arity 0. Equality, on the other hand, may not be included in c.

[4]That is to say, if c is a function constant then vc should be a function variable of the same arity; if c is a predicate constant then vc should be a predicate variable of the same arity.

and let $\upsilon\mathbf{c}$ stand for the list of all these variables. By $T^{\dagger}(\upsilon\mathbf{c})$ we denote the conjunction of the formulas

$$\forall\mathbf{x}(G \rightarrow F^{\mathbf{c}}_{\upsilon\mathbf{c}}) \tag{2.2}$$

for all rules $F \Leftarrow G$ of T, where $\mathbf{x}$ is the list of all free variables of F, G. (The expression $F^{\mathbf{c}}_{\upsilon\mathbf{c}}$ expresses that the result of substituting the variables $\upsilon\mathbf{c}$ for the corresponding constants $\mathbf{c}$ in F.) We view T as shorthand for the sentence

$$\forall\upsilon\mathbf{c}(T^{\dagger}(\upsilon\mathbf{c}) \leftrightarrow (\upsilon\mathbf{c} = \mathbf{c})). \tag{2.3}$$

(By $\upsilon\mathbf{c} = \mathbf{c}$ we denote the conjunction of the formulas $\upsilon c = c$ for all members c of the tuple $\mathbf{c}$.) Accordingly, by a model of the causal theory T we understand a model of (2.3) in the sense of classical logic. The models of T are characterized, informally speaking, by the fact that the interpretation of the explainable symbols $\mathbf{c}$ in the model is the only interpretation of these symbols that is "causally explained" by the rules of T.

In Section 2.3.5 we will show how the dynamic domain from Example 2.2.1 and 2.2.2 above can be described in causal logic.

2.3.2 Literal Completion

Let T be a causal theory such that all its explainable symbols are predicate constants. We say that T is *definite* if the heads of its rules are literals or do not contain explainable symbols. Literal completion introduced in [44] allows us to turn a definite causal theory into equivalent first-order sentence. In this review, we impose a more restrictive condition, similar to the definition of Clark normal form in [20, Section 6.1]. This is not a significant limitation, because any definite causal theory can be converted to this normal form by equivalent transformations.

We say that T is in *Clark normal form* if all its explainable symbols are predicate symbols, and it consists of

- rules of the form
$$p(\mathbf{x}) \Leftarrow G(\mathbf{x}), \tag{2.4}$$
one for each explainable predicate symbol p, where $\mathbf{x}$ is a tuple of distinct variables, and $G(\mathbf{x})$ is a formula without any free variables other than the members of $\mathbf{x}$,

- rules of the form
$$\neg p(\mathbf{x}) \Leftarrow G(\mathbf{x}), \tag{2.5}$$
one for each explainable predicate symbol p, where $\mathbf{x}$ and $G(\mathbf{x})$ are as above, and

- rules without explainable symbols in the head.

The *literal completion* of a causal theory T in Clark normal form is the conjunction of the sentences

$$\forall \mathbf{x}(p(\mathbf{x}) \leftrightarrow G(\mathbf{x})) \tag{2.6}$$

for all rules of T of the form (2.4), the sentences

$$\forall \mathbf{x}(\neg p(\mathbf{x}) \leftrightarrow G(\mathbf{x})) \tag{2.7}$$

for all rules of T of the form (2.5), and the sentences

$$\widetilde{\forall}(G \rightarrow F) \tag{2.8}$$

(the symbol $\widetilde{\forall}$ expresses that universal closure) for all rules $F \Leftarrow G$ of T without explainable symbols in the head.

Fact 2.3.1. *[44, Completion Theorem] Any causal theory in Clark normal form is equivalent to its literal completion.*

2.3.3 Examples

Example 2.3.1. Consider the propositional causal theory T_0

$$\begin{array}{r} p \Leftarrow \neg q \\ \neg q \Leftarrow p \end{array} \tag{2.9}$$

where p and q are explainable. According to the semantics of causal logic, T_0 is shorthand for the sentence

$$\forall \upsilon p \forall \upsilon q((\neg q \rightarrow \upsilon p) \wedge (p \rightarrow \neg \upsilon q) \leftrightarrow \upsilon p = p \wedge \upsilon q = q)).$$

The literal completion of T_0 is the formula

$$(p \leftrightarrow \neg q) \wedge (\neg p \leftrightarrow \bot) \wedge (\neg q \leftrightarrow p) \wedge (q \leftrightarrow \bot)$$

which is equivalent to $p \wedge \neg q$.

Example 2.3.2. Let T_1 be the causal theory consisting of two rules:

$$p(a) \Leftarrow \top$$

(the symbols $\bot$ and $\top$ denote the logical constant *false* and *true*) and

$$\neg p(x) \Leftarrow \neg p(x),$$

with the explainable symbol p. The first rule says that there is a cause for a to have property p. The second rule says that if an object does not have property p then there is a cause for that; including this rule in a causal theory has, informally speaking, the same effect as saying that p is false by default [44, Section 3]. In this case, $T^\dagger(\upsilon p)$ is

$$\upsilon p(a) \wedge \forall x(\neg p(x) \rightarrow \neg \upsilon p(x)),$$

so that T is understood as shorthand for the sentence

$$\forall \upsilon p(\upsilon p(a) \wedge \forall x(\neg p(x) \rightarrow \neg \upsilon p(x)) \leftrightarrow \forall x(\upsilon p(x) \leftrightarrow p(x))).$$

The literal completion of this theory is

$$\forall x(p(x) \leftrightarrow x = a) \wedge \forall x(\neg p(x) \leftrightarrow \neg p(x)). \tag{2.10}$$

Example 2.3.3. Causal theory T_2 has two rules:

$$\begin{array}{r} p(x) \Leftarrow q(x), \\ \neg p(x) \Leftarrow \neg p(x), \end{array} \tag{2.11}$$

and the predicate constant p is explainable. According to the semantics of causal logic, T_1 is shorthand for the sentence

$$\forall \upsilon p(\forall x(q(x) \rightarrow \upsilon p(x)) \wedge \forall x(\neg p(x) \rightarrow \upsilon p(x)) \;\leftrightarrow\; \upsilon p = p),$$

where υp is a predicate variable. Its literal completion, in simplified form, is

$$\forall x(p(x) \leftrightarrow q(x)).$$

2.3.4 Explainable Function Symbols

Recall that in a first-order causal theory, function symbols (in particular, object constants) can be explainable as well.

Example 2.3.4. Causal theory T_3 has the rules

$$\begin{aligned} \bot &\Leftarrow a = b, \\ c = a &\Leftarrow c = a, \\ c = b &\Leftarrow q, \end{aligned}$$

and the object constant c is explainable. The first rule of T_3 says that a is different from b. The second rule ("if $c = a$ then there is a cause for this") expresses, in the language of causal logic, that by default $c = a$. The last rule says that there is a cause for c to be equal to b if q is true. Theory T_3 is shorthand for the sentence

$$\forall vc((a = b \to \bot) \wedge (c = a \to vc = a) \wedge (q \to vc = b) \ \leftrightarrow \ vc = c)$$

where vc is an object variable. This formula is equivalent to

$$a \neq b \wedge (q \to c = b) \wedge (\neg q \to c = a). \tag{2.12}$$

The second conjunctive term shows that if q holds then the value of c is different from its default value a.

However, this theory is not definite, because of its explainable function symbol c. The process of literal completion is not applicable to it. In Chapter 3 we will generalize the definition of literal completion to allow explainable function symbols, which will allow us to use a procedure similar to the one introduced in [44] to turn causal theories such as T_3 into a first-order sentence.

We will also return to this example in Chapter 5 to illustrate the process of eliminating explainable functions in favor of explainable predicates.

2.3.5 Describing Dynamic Domains in Causal Logic

Example 2.3.5. Consider Example 2.2.1 (toggling switches). For simplicity we will only consider the time instants 0,1 and the execution of the toggle action at time 0. We will write $on_i(x)$ to express that switch x is on at time i, and $toggle(x)$ to express that switch x is toggled at time 0. This dynamic domain can be formalized by the following causal rules:

$$\begin{aligned} on_1(x) &\Leftarrow toggle(x) \land \neg on_0(x), \\ \neg on_1(x) &\Leftarrow toggle(x) \land on_0(x), \\ on_1(x) &\Leftarrow on_0(x) \land on_1(x), \\ \neg on_1(x) &\Leftarrow \neg on_0(x) \land \neg on_1(x). \end{aligned} \tag{2.13}$$

The first pair of rules describes the effect of toggling a switch x: this action causes the fluent $on(x)$ at time 1 to take the value opposite to its value at time 0. The second pair solves the frame problem for the fluent $on(x)$ by postulating that if the value of that fluent at time 1 is equal to its previous value then there is a cause for this. Inertia, in the sense of commonsense reasoning, is the cause, and these two rules represents commonsense law of inertia.The predicate symbol on_1 is the only explainable symbol.

Using literal completion, we can check that (2.13) is equivalent to

$$\forall x(on_1(x) \leftrightarrow ((on_0(x) \land \neg toggle(x)) \lor (\neg on_0(x) \land toggle(x)))). \tag{2.14}$$

In Chapter 4 we will see how this causal theory can be translated into the language of answer set programming, and use the stable model semantics to characterize its model which is equivalent to (2.14).

We will discuss also the elaboration of this example that contains, in addition to (2.13), the rule

$$dark \leftrightarrow \neg on_1(myswitch) \Leftarrow \top. \tag{2.15}$$

This is a nondefinite "synonymy" rule, and we will be able to translate rules like this into ASP as well.

Example 2.3.6. Consider Example 2.2.2 (moving objects) again, for the time instants 0,1. We would like to take into account the fact that the domain involves things of several kinds: movable objects, places, and time instants. To this end, we include the auxiliary symbol *none*, which is used as the value of $loc(x, t)$ when the arguments are "not of the right kind" (that is, when x is not a movable object or when t is not a time instant). The rules of the causal theory T_4 are

$$\begin{aligned}
\bot &\Leftarrow 0 = 1,\\
\bot &\Leftarrow 0 = none,\\
\bot &\Leftarrow 1 = none,\\
obj(x) \wedge place(y) &\Leftarrow move(x, y),\\
loc(x, 0) = y &\Leftarrow loc(x, 0) = y \wedge obj(x) \wedge place(y),\\
loc(x, 1) = y &\Leftarrow move(x, y),\\
loc(x, 1) = y &\Leftarrow loc(x, 0) = y \wedge loc(x, 1) = y \wedge obj(x) \wedge place(y),\\
loc(x, t) = none &\Leftarrow \neg obj(x),\\
loc(x, t) = none &\Leftarrow t \neq 0 \wedge t \neq 1,
\end{aligned}$$

and the function constant *loc* is explainable. The rule with $loc(x, 0)$ in the head allows an object x to be initially anywhere: whichever place is the value of $loc(x, 0)$, there is a cause for that. The next two rules describe the effect of moving objects and the inertia property of locations. According to the semantics of causal logic, T_4 is shorthand for the second-order formula

$$\forall \upsilon loc(T_4^\dagger(\upsilon loc) \leftrightarrow (\upsilon loc = loc)),$$

where υloc is a binary function variable.

Due to the presence of explainable function symbol *loc*, it is not possible to apply literal completion to transform it into an equivalent first-order sentence. In Chapter 3 we will accomplish that using the generalized literal completion.

To show that computing the model of T_4 can be automated, we will, in Chapter 5, eliminate explainable function symbols *loc* in favor of an auxiliary explainable predicate symbol *at*, thus turn this causal theory into a definite causal theory. This will not only allow us to use literal completion to characterize its models, but will also pave the way to turning it into a logic program, by the translation proposed in Section 4.3. Executable code shown in Section 5.6.2 shows that a model of T_4 can be computed by calling answer set solvers.

2.3.6 Disjoint Causal Theories

About causal theories T_1, T_2 with sets $\mathbf{c}_1$, $\mathbf{c}_2$ of explainable symbols we say that they are *disjoint* if

- $\mathbf{c}_1$ is disjoint from $\mathbf{c}_2$, and

- the symbols in $\mathbf{c}_1$ do not occur in the heads of the rules of T_2, and the symbols in $\mathbf{c}_2$ do not occur in the heads of the rules of T_1

[44, Section 6]. For any pairwise disjoint causal theories $T_1, \dots, T_m$, define their *union* to be the causal theory obtained by combining their rules and their explainable symbols.

Fact 2.3.2. *[44, Lemma 1] The union of pairwise disjoint causal theories $T_1, \dots, T_m$ is equivalent to the conjunction $T_1 \wedge \dots \wedge T_m$.*

2.4 Review of Stable Models

2.4.1 Definition of Stable Models

We adopt the view that first-order formulas are formed using the propositional connectives:

$$\top,\ \bot,\ \neg,\ \wedge,\ \vee,\ \rightarrow$$

(as well as the quantifiers $\forall$, $\exists$). The symbol $\neg$ corresponds to negation as failure ("`not`" in ASP programs) and not to strong (classical) negation in the sense of [28].

Stable models are only defined here for sentences of a special syntactic form. A first-order sentence is a *rule*[5] if it has the form $\widetilde{\forall}(F \rightarrow G)$ and has no occurrences of $\rightarrow$ other than the one explicitly shown. This expression can be abbreviated as $G \leftarrow F$. Rules of the form $G \leftarrow \top$ will be abbreviated as G.

[5]Or *program rule*, to distinguish it from causal rules in the sense of Section 2.3.1.

Rules of the form $\bot \leftarrow F$ will be abbreviated as $\leftarrow F$; such rules are called *constraints*. If the head of a rule has the form $p(\mathbf{t}) \vee \neg p(\mathbf{t})$, then it can be abbreviated as $\{p(\mathbf{t})\}$. Such rules are called *choice rules*. A *logic program* is a conjunction of rules.

The definition of a stable model below is more limited than the definition from [20] because it is only applicable to programs, not to arbitrary sentences. For instance, it does not cover the formulas $(p \rightarrow q) \rightarrow r$ and $(p \rightarrow q) \vee r$. On the other hand, it is simpler than the general definition, and it is sufficient for this dissertation.

We need the following notation from [43]. If p and q are predicate constants of the same arity then $p \leq q$ stands for the formula

$$\forall \mathbf{x}(p(\mathbf{x}) \rightarrow q(\mathbf{x})),$$

where $\mathbf{x}$ is a tuple of distinct object variables. If $\mathbf{p}$ and $\mathbf{q}$ are tuples $p_1, \ldots, p_n$ and $q_1, \ldots, q_n$ of predicate constants then $\mathbf{p} \leq \mathbf{q}$ stands for the conjunction

$$(p_1 \leq q_1) \wedge \cdots \wedge (p_n \leq q_n),$$

and $\mathbf{p} < \mathbf{q}$ stands for $(\mathbf{p} \leq \mathbf{q}) \wedge \neg(\mathbf{q} \leq \mathbf{p})$. In second-order logic, we apply the same notation to tuples of predicate variables.

Let $\mathbf{p}$ be a list of distinct predicate constants; members of $\mathbf{p}$ will be called *intensional predicates*.[6] For each $p \in \mathbf{p}$, choose a predicate variable υp

[6]This list usually consists of all predicate symbols occurring in the heads of rules; those are the predicates that we "intend to characterize" by the rules of the program.

of the same arity, and let $\upsilon\mathbf{p}$ stand for the list of all these variables. For any logic program F, by $\mathrm{SM}_{\mathbf{p}}[F]$ we denote the second-order sentence

$$F \wedge \neg\exists\upsilon\mathbf{p}((\upsilon\mathbf{p} < \mathbf{p}) \wedge F^{\diamond}(\upsilon\mathbf{p})), \tag{2.16}$$

where $F^{\diamond}(\upsilon\mathbf{p})$ is the formula obtained from F by replacing, for every $p \in \mathbf{p}$, each occurrence of p that is not in the scope of negation with υp. A model of F is *stable* (relative to the set $\mathbf{p}$ of intensional predicates) if it satisfies $\mathrm{SM}_{\mathbf{p}}[F]$.

Example 2.4.1. Let F be the propositional formula $\neg p \to q$. If both p and q are intensional then $F^{\diamond}(\upsilon p, \upsilon q)$ is

$$\neg p \to \upsilon q,$$

so that $\mathrm{SM}_{pq}[F]$ is

$$(\neg p \to q) \wedge \neg\exists(\upsilon p)(\upsilon q)(((\upsilon p, \upsilon q) < (p, q)) \wedge (\neg p \to \upsilon q)).$$

This formula is equivalent to $\neg p \wedge q$. Consequently F has one stable model: p is false and q is true.

Example 2.4.2. Let F be the formula

$$\forall x(\neg p(x) \to (q(x) \vee \neg q(x))) \tag{2.17}$$

If we take q to be the only intensional predicate then $F^{\diamond}(\upsilon q)$ is

$$\forall x(\neg p(x) \to (\upsilon q(x) \vee \neg q(x))).$$

Consequently $\mathrm{SM}_q[F]$ is

$$\forall x(\neg p(x) \to (q(x) \vee \neg q(x))) \wedge \neg\exists\upsilon q((\upsilon q < q) \wedge \forall x(\neg p(x) \to (\upsilon q(x) \vee \neg q(x)))).$$

The first conjunctive term here is logically valid and can be dropped. The second is equivalent to the first-order formula $\neg\exists x(p(x) \land q(x))$, which reflects the intuitive meaning of choice: q is an arbitrary set disjoint from p.

If two formulas F and G are intuitionistically equivalent then they have the same stable models. Moreover, they are "strongly equivalent": for any formula H, $F \land H$ and $G \land H$ have the same stable models [20].

2.4.2 Describing Dynamic Domains by Logic Programs

In Section 2.3.5, we saw how moving objects can be described in causal logic. Now we will represent the same domain in logic programming.

Example 2.4.3. The signature of the logic program M consists of

- the object constants $\widehat{0}, \ldots, \widehat{k}$, where k is a fixed nonnegative integer;
- the unary predicate constants *object*, *place*, and *step*; they correspond to the three types of individuals under consideration;
- the binary predicate constant *next*; it describes the temporal order of steps;
- the ternary predicate constants *at* and *move*; they represent the fluents and actions that we are interested in.

The predicate constants *step*, *next*, and *at* are intensional; the other three are not. The program consists of the following rules:

(i) the facts

$$step(\widehat{0}),\ step(\widehat{1}),\ \dots\ step(\widehat{k});$$
$$next(\widehat{0},\widehat{1}),\ next(\widehat{1},\widehat{2}),\ \dots,\ next(\widehat{k-1},\widehat{k});$$

(ii) the unique name constraints

$$\leftarrow \widehat{i} = \widehat{j} \qquad (1 \le i < j \le k);$$

(iii) the constraints describing the arguments of *at* and *move*:

$$\leftarrow at(x,y,z) \wedge \neg(object(x) \wedge place(y) \wedge step(z))$$

and

$$\leftarrow move(x,y,z) \wedge \neg(object(x) \wedge place(y) \wedge step(z));$$

(iv) the uniqueness of location constraint

$$\leftarrow at(x,y_1,z) \wedge at(x,y_2,z) \wedge y_1 \neq y_2;$$

(v) the existence of location constraint

$$\leftarrow object(x) \wedge step(z) \wedge \neg\exists y\ at(x,y,z);$$

(vi) the rule expressing the effect of moving an object:

$$at(x,y,u) \leftarrow move(x,y,z) \wedge next(z,u);$$

(vii) the choice rule expressing that initially an object can be anywhere:

$$\{at(x,y,0)\} \leftarrow object(x) \wedge place(y);$$

(viii) the choice rule expressing the commonsense law of inertia:[7]

$$\{at(x,y,u)\} \leftarrow at(x,y,z) \wedge next(z,u).$$

In Chapter 6 we define an action language $\mathcal{BC}$ whose semantics is based on logic programming. We use it to describe dynamic domains: blocks world (Example 2.2.3) and leaking container (Example 2.2.4). By calling answer set solvers, reasoning about the dynamic domains can be automated.

2.4.3 Properties of Stable Models

Here are some general properties of stable models.

For a list $\mathbf{p}$ of predicate symbols, $\mathit{Choice}(\mathbf{p})$ stands for the conjunction of the choice formulas

$$\forall \mathbf{x}(p(\mathbf{x}) \vee \neg p(\mathbf{x})),$$

where $\mathbf{x}$ is a tuple of distinct variables, for all members p of list $\mathbf{p}$. In particular, if p is a propositional symbol then $\mathit{Choice}(p)$ stands for $p \vee \neg p$.

Fact 2.4.1. *For any program F and any disjoint lists $\boldsymbol{p}$, $\boldsymbol{q}$ of distinct predicate constants,*

(a) $\mathit{SM}[F;\boldsymbol{pq}]$ entails $\mathit{SM}[F;\boldsymbol{p}]$,

(b) $\mathit{SM}[F \wedge \mathit{Choice}(\boldsymbol{q});\boldsymbol{pq}]$ is equivalent to $\mathit{SM}[F;\boldsymbol{p}]$.

[7]This representation of inertia follows the example of [5, Figure 1].

Recall that an occurrence of a symbol in a formula is F *positive* if the number of implications containing that occurrence in the antecedent is even, and *strictly positive* if that number is 0. (Recall that we treat $\neg F$ as an abbreviation for the implication $F \to \bot$.) We say that an occurrence of a predicate constant in a formula is *negated* if it belongs to a subformula of the form $\neg F$ (that is, $F \to \bot$), and *nonnegated* otherwise. For instance, in the formula

$$p(x) \wedge \neg r(x) \to p(x) \tag{2.18}$$

both p and r are positive, p is strictly positive, and r is negated.

Fact 2.4.2. *For any programs F and G and any list $\boldsymbol{p}$ of predicate symbols,*

(a) $SM[F;\boldsymbol{p}] \wedge G$ entails $SM[F \wedge G;\boldsymbol{p}]$,

(b) if G does not contain strictly positive occurrences of symbols from $\boldsymbol{p}$ then $SM[F;\boldsymbol{p}] \wedge G$ is equivalent to $SM[F \wedge G;\boldsymbol{p}]$.

Assertion (a) is immediate from the definition of SM. Assertion (b) is proved in [20, Section 5.1].

Fact 2.4.3. *For any programs F and G and any list $\boldsymbol{p}$ of predicate symbols, if the equivalence $F \leftrightarrow G$ can be derived in intuitionistic logic from the formulas $Choice(q)$ for the predicate symbols q that do not belong to $\boldsymbol{p}$ then $SM[F;\boldsymbol{p}]$ is equivalent to $SM[G;\boldsymbol{p}]$.*

This is a weaker form of [20, Theorem 5].

In the following we introduce the splitting theorem from [21]. For a first-order formula of F, the *predicate dependency graph of* F relative to the list of $\mathbf{p}$ of intensional predicates, is the directed graph that

- has all intensional predicates as its vertices, and
- has an edge from p to q, for some rule $G \rightarrow H$ of F,
  - p has a strictly positive occurrence in H, and
  - q has a positive nonnegated occurrence in G.

For instance, the predicate dependency graph of (2.18) relative to p, q, r has one edge, from p to q. We will denote the predicate dependency graph of F relative to $\mathbf{p}$ by $DG_{\mathbf{p}}[F]$.

About a formula F we say that it is *negative* on a tuple $\mathbf{p}$ of predicate constants if members of $\mathbf{p}$ have no strictly positive occurrences in F.

Fact 2.4.4. *[21, Splitting Theorem] Let F, G be programs, and let $\boldsymbol{p}$, $\boldsymbol{q}$ be disjoint tuples of distinct predicate constants. If*

- *each strongly connected component of $DG_{\boldsymbol{pq}}[F \wedge G]$ is a subset of $\boldsymbol{p}$ or a subset of $\boldsymbol{q}$,*
- *F is negative on $\boldsymbol{q}$, and*
- *G is negative on $\boldsymbol{p}$,*

then $SM_{\boldsymbol{pq}}[F \wedge G]$ is equivalent to

$$SM_{\boldsymbol{p}}[F] \wedge SM_{\boldsymbol{q}}[G].$$

2.4.4 Lloyd-Topor Programs and Completion

The process of completing a logic program was introduced in [11] and generalized in [56]. A *Lloyd-Topor program* is a program consisting of rules of the form

$$p(\mathbf{t}) \leftarrow G, \tag{2.19}$$

where $\mathbf{t}$ is a tuple of terms, and G is a formula.

Let Π be a Lloyd-Topor program, and p a predicate constant (other than equality). Let

$$p(\mathbf{t}^i) \leftarrow G_i \qquad (i = 1, 2, \dots) \tag{2.20}$$

be all rules of Π that contain p in the head. The *definition of* p *in* Π is the rule

$$p(\mathbf{x}) \leftarrow \bigvee_i \exists \mathbf{y}^i (\mathbf{x} = \mathbf{t}^i \wedge G_i), \tag{2.21}$$

where $\mathbf{x}$ is a list of distinct variables not appearing in any of the rules (2.20), and $\mathbf{y}^i$ is the list of free variables of (2.20).[8] The *completed definition of* p *in* Π is the formula

$$\forall \mathbf{x} \left(p(\mathbf{x}) \leftrightarrow \bigvee_i \exists \mathbf{y}^i (\mathbf{x} = \mathbf{t}^i \wedge G_i) \right). \tag{2.22}$$

For instance, the completed definitions of p and q in program

$$\begin{array}{l} p(a), \\ q(b), \\ p(x) \leftarrow q(x) \end{array} \tag{2.23}$$

[8]By $\mathbf{x} = \mathbf{t}^i$ we denote the conjunction of the equalities between members of the tuple $\mathbf{x}$ and the corresponding members of the tuple $\mathbf{t}^i$.

are the formulas

$$\begin{array}{l}\forall x_1(p(x_1) \leftrightarrow x_1 = a \vee \exists x(x_1 = x \wedge q(x))),\\ \forall x_1(q(x_1) \leftrightarrow x_1 = b),\end{array}$$

which can be equivalently rewritten as

$$\begin{array}{l}\forall x(p(x) \leftrightarrow x = a \vee q(x)),\\ \forall x(q(x) \leftrightarrow x = b).\end{array} \tag{2.24}$$

By Comp[Π] we denote the conjunction of the completed definitions of all predicate constants p in Π. This sentence is similar to the completion of Π in the sense of [57, Section 2], except that it does not include Clark equality axioms.

Program completion can be used to characterize the stable models of logic programs, if they are "tight." We will review now the definition of tightness from [20, Section 7.3]. In application to a Lloyd-Topor program Π, when all predicate constants occurring in Π are treated as intensional, that definition can be stated as follows.

The *predicate dependency graph of* Π is the directed graph that has

- all predicate constants occurring in Π as its vertices, and
- an edge from p to q whenever Π contains a rule (2.19) with p in the head such that its body G has a positive nonnegated occurrence of q.

We say that Π is *tight* if the predicate dependency graph of Π is acyclic.

For example, the predicate dependency graph of program (2.23) has a single edge, from p to q. The predicate dependency graph of program

$$\begin{array}{l} p(x) \leftarrow q(x), \\ q(a) \leftarrow p(b). \end{array} \tag{2.25}$$

has two edges, from p to q and from q to p. The predicate dependency graph of the program

$$\begin{array}{l} p(a,b) \\ q(x,y) \leftarrow p(y,x) \wedge \neg p(x,y) \end{array} \tag{2.26}$$

has a single edge, from q to p (because one of the occurrences of p in the body of the second rule is nonnegated). The predicate dependency graph of the program

$$\begin{array}{l} p(x) \leftarrow q(x), \\ q(x) \leftarrow r(x), \\ r(x) \leftarrow s(x) \end{array} \tag{2.27}$$

has 3 edges:

$$p \longrightarrow q \longrightarrow r \longrightarrow s.$$

Programs (2.23), (2.26) and (2.27) are tight; program (2.25) is not.

Fact 2.4.5. *If a Lloyd-Topor program* Π *is tight then* $SM[\Pi]$ *is equivalent to* $Comp[\Pi]$.

This is an easy corollary to a theorem from [20]. Indeed, consider the set Π' of the definitions (2.21) of all predicate constants p in Π. It can be viewed as a formula in Clark normal form in the sense of [20, Section 6.1]. It is tight, because it has the same predicate dependency graph as Π. By Theorem 11 from [20], SM[Π'] is equivalent to the completion of Π' in the sense of [20, Section 6.1], which is identical to Comp[Π]. It remains to observe that Π is intuitionistically equivalent to Π', so that SM[Π] is equivalent to SM[Π'] [20, Section 5.1].

To illustrate limitations of Fact 2.4.5, note that it cannot be used to characterize the stable models of the program M in Example 2.4.3 in terms of completion. There are two reasons: first, M contains choice rules and constraints, which are not allowed in a Lloyd-Topor program; second, M is not tight. In Chapter 8 we generalize the notion of tightness using a new concept of "rule dependency graph" instead of the less informative "predicate dependency graph." We also show how to use this concept to characterize the stable model of some programs describing dynamic domains such as M.

2.5 Strong Negation

In many ASP programs, we distinguish between two kinds of negation: negation as failure discussed above, and strong (classical) negation [32].

In this dissertation, we will refer to strong negation only in the context of propositional programs. We will distinguish between propositional atoms of two kinds, *positive* and *negative*, and assume that each negative atom is an expression of the form $\sim A$, where A is a positive atom. The symbol $\sim$ represents strong negation. A stable model of a logic program with strong negation is called its *answer set* if it does not contain "complementary" pairs of atoms $A, \sim A$.

Answer set solvers can be used for generating answer sets of programs of two kinds of negation. In their input languages, negation as failure is denoted by `not`, and strong negation is denoted by `-`.

Chapter 3

Functional Completion

Literal completion introduced in Section 2.3.2 is applicable to a theory only if each of its explainable symbols is a predicate constant; function constants are allowed in the signature, but they cannot be explainable. As we mentioned in the Introduction, explainable function symbols are often useful. It is possible, of course, to replace the function symbol *loc* in Example 2.3.6 by the predicate symbol *at* as in Example 2.4.3, but it would make the representation less concise. The advantages of using functional notation in such cases are the same as the advantages of writing $x + y = z$ in formal arithmetic in comparison with $sum(x, y, z)$: there is no need to postulate the existence and uniqueness of the value of the function, and many ideas can be expressed more concisely. For instance, we can write

$$loc(x_1, t) = loc(x_2, t)$$

instead of

$$\exists y(at(x_1, y, t) \wedge at(x_2, y, t)).$$

Our goal here is to extend the definition of literal completion and the theorem on literal completion reviewed above (Fact (2.3.1)) to causal theories with explainable function symbols.

3.1 Clark Normal Form Extended to Explainable Functions

The definition of Clark normal form in Section 2.3.2 is extended to causal theories with explainable functions by adding an extra clause. About a causal theory T we say that it is in *Clark normal form* if it consists of

- rules of the form (2.4), one for each explainable predicate symbol p,
- rules of the form (2.5), one for each explainable predicate symbol p,
- rules of the form
$$f(\mathbf{x}) = y \Leftarrow G(\mathbf{x}, y), \tag{3.1}$$
one for each explainable function symbol f, where $\mathbf{x}, y$ is a tuple of distinct variables, and $G(\mathbf{x}, y)$ is a formula without any free variables other than the members of $\mathbf{x}, y$,
- rules without explainable symbols in the head.

In many cases, a causal theory can be transformed into an equivalent causal theory in Clark normal form. For instance, the causal theory T_3 in Example 2.3.4 can be converted to Clark normal form by rewriting its last two rules as

$$\begin{array}{l} c = x \Leftarrow x = a \wedge c = a, \\ c = x \Leftarrow x = b \wedge q \end{array}$$

and then merging them into one rule:

$$c = x \Leftarrow (x = a \wedge c = a) \vee (x = b \wedge q). \tag{3.2}$$

It is clear that the part of $T_3^\dagger(vc)$ contributed by the last two rules of T_3 is logically equivalent to the part contributed by (3.2). Similarly, the Clark normal form of T_4 in Example 2.3.6 is

$$\begin{aligned}
\bot &\Leftarrow 0 = 1, \\
\bot &\Leftarrow 0 = none, \\
\bot &\Leftarrow 1 = none, \\
obj(x) \wedge place(y) &\Leftarrow move(x,y), \\
loc(x,t) = y &\Leftarrow (t = 0 \wedge loc(x,0) = y \wedge obj(x) \wedge place(y)) \\
&\quad \vee (t = 1 \wedge move(x,y)) \\
&\quad \vee (t = 1 \wedge loc(x,0) = y \wedge loc(x,1) = y \\
&\qquad \wedge obj(x) \wedge place(y)) \\
&\quad \vee (y = none \wedge \neg obj(x)) \\
&\quad \vee (y = none \wedge t \neq 0 \wedge t \neq 1).
\end{aligned} \tag{3.3}$$

3.2 Literal Completion Extended to Explainable Functions

Functional completion is a generalization of literal completion to causal theories in Clark normal form that may include explainable functions. The *functional completion* of a causal theory T in Clark normal form is the conjunction of

- sentences (2.6) for all rules of T of the form (2.4),
- sentences (2.7) for all rules of T of the form (2.5),
- sentences

  $$\widetilde{\forall}(f(\mathbf{x}) = y \leftrightarrow G(\mathbf{x}, y))$$

  for all rules of T of the form (3.1), and

- sentences (2.8) for all rules $F \Leftarrow G$ of T without explainable symbols in the heads.

We will denote the functional completion of T by Comp$[T]$.

Theorem 3.2.1. *For any causal theory T in Clark normal form,*

$$\exists x_1 x_2 (x_1 \neq x_2) \tag{3.4}$$

entails $T \leftrightarrow Comp[T]$.

Corollary 3.2.2. *If a causal theory in Clark normal form contains a rule of the form $\bot \Leftarrow t_1 = t_2$ then it is equivalent to its functional completion.*

Consider, for instance, theory T_3 in Example 2.3.4. As discussed above, its Clark normal form consists of rules (3.2) and

$$\bot \Leftarrow a = b.$$

Its functional completion is the conjunction of the formulas

$$\forall x (c = x \leftrightarrow (x = a \wedge c = a) \vee (x = b \wedge q))$$

and $a = b \rightarrow \bot$ (that is, $a \neq b$). By the corollary, this conjunction is equivalent to T_3.

The Clark normal form of T_4 is (3.3). The functional completion of this theory is the conjunction of the formulas

$$\begin{array}{l}
\qquad\qquad\quad 0 \neq 1, \quad 0 \neq \mathit{none}, \quad 1 \neq \mathit{none},\\
\qquad\quad \forall xy(\mathit{move}(x,y) \rightarrow \mathit{obj}(x) \wedge \mathit{place}(y)),\\
\forall xty(\mathit{loc}(x,t) = y \leftrightarrow (t = 0 \wedge \mathit{loc}(x,0) = y \wedge \mathit{obj}(x) \wedge \mathit{place}(y))\\
\qquad\qquad \vee(t = 1 \wedge \mathit{move}(x,y))\\
\qquad\qquad \vee(t = 1 \wedge \mathit{loc}(x,0) = y \wedge \mathit{loc}(x,1) = y\\
\qquad\qquad\quad \wedge \mathit{obj}(x) \wedge \mathit{place}(y))\\
\qquad\qquad \vee(y = \mathit{none} \wedge \neg \mathit{obj}(x))\\
\qquad\qquad \vee(y = \mathit{none} \wedge t \neq 0 \wedge t \neq 1)).
\end{array}$$

By the corollary, this conjunction is equivalent to T_4. Using equivalent transformations in first-order-logic, we can rewrite it as the conjunction of the formulas

$$\begin{array}{l}
\qquad\qquad\quad 0 \neq 1, \quad 0 \neq \mathit{none}, \quad 1 \neq \mathit{none},\\
\qquad\quad \forall xy(\mathit{move}(x,y) \rightarrow \mathit{obj}(x) \wedge \mathit{place}(y)),\\
\qquad\quad \forall x(\mathit{obj}(x) \rightarrow \mathit{place}(\mathit{loc}(x,0))),\\
\quad \forall xt((\neg \mathit{obj}(x) \vee (t \neq 0 \wedge t \neq 1)) \rightarrow \mathit{loc}(x,t) = \mathit{none}),\\
\forall xy(\mathit{obj}(x) \rightarrow\\
\quad \mathit{loc}(x,1) = y \leftrightarrow (\mathit{move}(x,y) \vee (\mathit{loc}(x,0) = y \wedge \neg \exists w\ \mathit{move}(x,w)))).
\end{array}$$

The last of these formulas characterizes the location of an object at time 1 in terms of its location at time 0 and the actions that have been executed. In this sense, it is similar to successor state axioms as defined in [69].

Without the assumption that the theory contains a rule $\bot \Leftarrow t_1 = t_2$ the assertion of the corollary would be incorrect. For instance, consider the causal theory consisting of one rule

$$c = x \Leftarrow \bot,$$

where c is an explainable object constant. This theory is equivalent to

$$\forall \upsilon c(\upsilon c = c);$$

its completion is equivalent to $\bot$.

3.3 Proof of Theorem 3.2.1

3.3.1 A Special Case

We will first prove the theorem from Section 3.2 for the special case when T consists of a single rule (3.1), where f is explainable. We need to show that (3.4) entails the equivalence between

$$\forall \upsilon f(\forall \mathbf{x} y(G(\mathbf{x}, y) \rightarrow \upsilon f(\mathbf{x}) = y) \leftrightarrow \upsilon f = f) \tag{3.5}$$

and

$$\forall \mathbf{x} y(f(\mathbf{x}) = y \leftrightarrow G(\mathbf{x}, y)). \tag{3.6}$$

Right-to-left: under assumption (3.6), formula (3.5) is equivalent to the logically valid formula

$$\forall \upsilon f(\forall \mathbf{x} y(f(\mathbf{x}) = y \rightarrow \upsilon f(\mathbf{x}) = y) \leftrightarrow \upsilon f = f).$$

Left-to-right: assume (3.5), that is,

$$\forall \upsilon f(\forall \mathbf{x} y(G(\mathbf{x}, y) \rightarrow \upsilon f(\mathbf{x}) = y) \rightarrow \upsilon f = f) \tag{3.7}$$

and

$$\forall \mathbf{x} y(G(\mathbf{x}, y) \rightarrow f(\mathbf{x}) = y). \tag{3.8}$$

The last formula is one half of equivalence (3.6). It remains to derive the other half, that is, $G(\mathbf{x}, f(\mathbf{x}))$. Assume that for some $\mathbf{x}^0$, $\neg G(\mathbf{x}^0, f(\mathbf{x}^0))$. By

(3.4), there exists a y_0 different from $f(\mathbf{x}^0)$. We will prove that the function υf defined by the condition

$$\upsilon f(\mathbf{x}^0) = y_0 \wedge \forall \mathbf{x}(\mathbf{x} \neq \mathbf{x}^0 \to \upsilon f(\mathbf{x}) = f(\mathbf{x}))$$

satisfies the antecedent of (3.7). Assume $G(\mathbf{x}, y)$. Since $\neg G(\mathbf{x}^0, y)$, $\mathbf{x} \neq \mathbf{x}_0$. Then $\upsilon f(\mathbf{x}) = f(\mathbf{x})$. On the other hand, by (3.8), $f(\mathbf{x}) = y$. Consequently $\upsilon f(\mathbf{x}) = y$; the antecedent of (3.7) is proved. It follows that the consequent $\upsilon f = f$ holds, so that $y_0 = \upsilon f(\mathbf{x}^0) = f(\mathbf{x}^0)$. This is impossible by the choice of y_0.

3.3.2 The General Case

Let T be a causal theory in Clark normal form, and let $f_1, \dots, f_m$ be its explainable function symbols. For each $i = 1, \dots, m$, let T_i be the causal theory whose only rule is the rule of T that contains f_i in the head, with f_i as its only explainable symbol. Let T_{m+1} be the causal theory whose rules are the rules of T that do not contain explainable function symbols in their heads, and whose set of explainable symbols is the set of all explainable predicate symbols of T. It is clear that theories $T_1, \dots, T_m, T_{m+1}$ are pairwise disjoint, and that their union is T. By Fact 2.3.2, it follows that T is equivalent to $T_1 \wedge \dots T_m \wedge T_{m+1}$. According to the special case proved in Section 3.3.1, (3.4) entails

$$T_i \leftrightarrow \text{Comp}[T_i] \qquad (i = 1, \dots, m).$$

By Fact 2.3.1, T_{m+1} is equivalent to Comp$[T_{m+1}]$. Consequently (3.4) entails

$$T \leftrightarrow \text{Comp}[T_1] \wedge \cdots \wedge \text{Comp}[T_m] \wedge \text{Comp}[T_{m+1}].$$

It remains to observe that the right-hand side of this equivalence is Comp$[T]$.

To sum up, the process of completion, extended in this chapter to fluents represented by function symbols, allows us in some cases to turn a causal theory into an equivalent first-order formula. This possibility is important because, semantically, first-order languages are simpler and better understood than many nonmonotonic languages. The completion process is useful also because it clarifies the relationship between causal logic and monotonic solutions to the frame problem, such as those based on the approach of [69].

Chapter 4

Representing First-order Causal Theories in Logic Programming

In this chapter we will generalize McCain's translation [58] and Ferraris' translation [16] from causal theories to logic programming in several ways. First, we discard the requirement that the bodies of the given causal rules be conjunctions of literals. Second, instead of requiring that the head of each causal rule be a literal, we allow the heads to be disjunctions of literals. In this more general setting, the logic program corresponding to the given causal theory becomes disjunctive as well.

Third, we study causal rules with heads of the form $L_1 \leftrightarrow L_2$, where L_1 and L_2 are literals. Such a rule says that there is a cause for L_1 and L_2 to be equivalent ("synonymous") under some condition, expressed by the body of the rule. As discussed in Introduction, synonymy rules play an important role in the theory of commonsense reasoning in view of the fact that humans often explain the meaning of words by referring to their synonyms. A synonymy rule

$$L_1 \leftrightarrow L_2 \Leftarrow G \tag{4.1}$$

can be translated into logic programming by rewriting it as the pair of rules

$$\begin{array}{c} L_1 \vee \overline{L_2} \Leftarrow G \\ \overline{L_1} \vee L_2 \Leftarrow G \end{array}$$

where $\overline{L}$ expresses that the literal complementary to L, and then using our extension of McCain's translation to rules with disjunctive heads. It turns out, however, that there is no need to use disjunctive logic programs in the case of synonymy rules. If, for instance, G in (4.1) is an atom then the following group of nondisjunctive rules with strong negation will do:

$$\begin{array}{l} (L_1)^{\neg}_{\sim} \leftarrow (L_2)^{\neg}_{\sim} \wedge \neg \sim G \\ (L_2)^{\neg}_{\sim} \leftarrow (L_1)^{\neg}_{\sim} \wedge \neg \sim G \\ (\overline{L_1})^{\neg}_{\sim} \leftarrow (\overline{L_2})^{\neg}_{\sim} \wedge \neg \sim G \\ (\overline{L_2})^{\neg}_{\sim} \leftarrow (\overline{L_1})^{\neg}_{\sim} \wedge \neg \sim G. \end{array}$$

where $L^{\neg}_{\sim}$ expresses that the atom obtained by replacing $\neg$ in an literal L by $\sim$.

Finally, we extend the translation from propositional causal rules to first-order causal rules in the sense of [44]. This version of causal logic is useful for defining the semantics of variables in action descriptions [48].

4.1 McCain's Translation Revisited

In this section we assume that a causal theory T is propositional and all its atoms are explainable.

4.1.1 Basic McCain's Translation

McCain's translation is applicable to a causal theory T if the head of each rule of T is a literal, and the body is a conjunction of literals:

$$L \Leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \neg A_n. \tag{4.2}$$

The corresponding logic program consists of the logic programming rules

$$L^{\neg}_{\sim} \leftarrow \neg \sim A_1 \wedge \cdots \wedge \neg \sim A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \tag{4.3}$$

for all rules (4.2) of T. According to Proposition 6.7 from [58], complete answer sets of this logic program are identical to the models of T. (A set of atoms is *complete* if it contains exactly one member of each complementary pair of atoms $A, \sim A$. We identify a complete set of atoms with the corresponding truth assignment.)

For instance, McCain's translation turns causal theory T_0 in Example 2.3.1 into

$$\begin{array}{rl} p & \leftarrow \neg q \\ \sim q & \leftarrow \neg \sim p. \end{array}$$

The only answer set of this program is $\{p, \sim q\}$. It is complete, and it corresponds to the model of causal theory T_0.

4.1.2 Incorporating Constraints

In causal logic, a *constraint* is a rule with the head $\bot$ (falsity). McCain's translation can be easily extended to constraints with a conjunction of literals in the body—causal rules of the form

$$\bot \Leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n. \tag{4.4}$$

In the language of logic programming, (4.4) can be represented by a rule similar to (4.3):

$$\bot \leftarrow \neg \sim A_1 \wedge \cdots \wedge \neg \sim A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n. \tag{4.5}$$

Furthermore, each of the combinations $\neg\, \sim$ in (4.5) can be dropped without destroying the validity of the translation; that is to say, the rule

$$\bot \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \tag{4.6}$$

can be used instead of (4.5).

4.1.3 Eliminating Strong Negation

A negative atom $\sim A$ can be replaced in a logic program by a (regular) atom $\widehat{A}$ if we add a constraint that doesn't allow A and $\widehat{A}$ to be in the same stable model. In this way, we arrive at the *modified McCain's translation* of a causal theory T that does not use strong negation.

The modified McCain translation of a causal theory consisting of rules of the forms (4.2) and (4.4) includes

- rules (4.6) corresponding to constraints (4.4);
- rules corresponding to rules (4.2):

$$A_0 \leftarrow \neg \widehat{A_1} \wedge \cdots \wedge \neg \widehat{A_m} \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \tag{4.7}$$

if L is a positive literal A_0, and

$$\widehat{A_0} \leftarrow \neg\widehat{A_1} \wedge \cdots \wedge \neg\widehat{A_m} \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \tag{4.8}$$

if L is a negative literal $\neg A_0$;

- the constraints

$$\begin{array}{l} \leftarrow A \wedge \widehat{A} \\ \leftarrow \neg A \wedge \neg\widehat{A} \end{array} \tag{4.9}$$

for all atoms A.

For instance, the modified McCain translation of T_0 in Example 2.3.1 is

$$\begin{array}{ll} p & \leftarrow \neg q \\ \widehat{q} & \leftarrow \neg\widehat{p} \\ & \leftarrow p \wedge \widehat{p} \\ & \leftarrow \neg p \wedge \neg\widehat{p} \\ & \leftarrow q \wedge \widehat{q} \\ & \leftarrow \neg q \wedge \neg\widehat{q}. \end{array} \tag{4.10}$$

The only stable model of this program is $\{p, \widehat{q}\}$.

4.1.4 Translating Arbitrary Definite Theories

The requirement, in the definition of McCain's translation, that the bodies of all causal rules should be conjunctions of literals can be lifted by slightly modifying the translation process. Take any set T of causal rules of the forms

$$A \Leftarrow G, \tag{4.11}$$

$$\neg A \Leftarrow G, \tag{4.12}$$

$$\bot \Leftarrow G, \tag{4.13}$$

where A is an atom and G is an arbitrary propositional formula. For each rule (4.11), take the formula $A \leftarrow \neg\neg G$; for each rule (4.12), the formula $\widehat{A} \leftarrow \neg\neg G$; for each rule (4.13), the formula $\neg G$. Then add completeness constraints

$$\begin{array}{l} \bot \leftarrow A \wedge \widehat{A} \\ \bot \leftarrow \neg A \wedge \neg \widehat{A}. \end{array} \tag{4.14}$$

for all atoms A. Stable models of this collection of propositional formulas correspond to the models of T.

In application to T_0 in Example 2.3.1, this modification of McCain's translation gives

$$\begin{array}{l} p \leftarrow \neg\neg\neg q \\ \widehat{q} \leftarrow \neg\neg p \\ \bot \leftarrow p \wedge \widehat{p} \\ \bot \leftarrow \neg p \wedge \neg \widehat{p} \\ \bot \leftarrow q \wedge \widehat{q} \\ \bot \leftarrow \neg q \wedge \neg \widehat{q}. \end{array} \tag{4.15}$$

It is not surprising that (4.15) has the same stable model as (4.10): the two collections of formulas are intuitionistically equivalent to each other.[1]

4.2 Four Types of Causal Rules

In the rest of the chapter, we assume that the bodies of causal rules do not contain implication. This is not an essential limitation, because in classical

[1]Indeed, $\neg\neg\neg q$ is intuitionistically equivalent to $\neg q$; the equivalence between $\neg\neg p$ and $\neg\widehat{p}$ is intuitionistically entailed by the formulas $\neg(p \wedge \widehat{p})$ and $\neg(\neg p \wedge \neg\widehat{p})$, which belong both to (4.10) and to (4.15).

logic $\rightarrow$ can be expressed in terms of other connectives, and the meaning of a causal rule does not change if we replace its body (or head) by a classically equivalent formula.

Here are four types of rules that we are going to consider, in the order of increasing complexity of their heads:

- The head is $\bot$, that is, the rule is a constraint. Such causal rules will be also called *C-rules*.
- The head is a literal containing an explainable predicate symbol. These are *L-rules*.
- The head has the form $L_1 \leftrightarrow L_2$, where each L_i is a literal containing an explainable predicate symbol. These are *synonymy rules*, or *S-rules*.
- The head has the form $L_1 \vee \cdots \vee L_n$ ($n \geq 0$), where each L_i is a literal containing an explainable predicate symbol. These are *D-rules*.

All C-rules and L-rules can be viewed also as D-rules, and any S-rule can be replaced with an equivalent pair of D-rules (see Lemma 4.8.11 in Section 4.8.2). Nevertheless, we give special attention here to rules of the first three types, and the reason is that our translation handles such rules in special ways. It appears that causal rules of types C, L, and S will be more important than general D-rules in applications of this work to the automation of reasoning about actions.

On the other hand, the possibility of reducing types C, L, and S to type D plays an important role in the proof of the soundness of our translation (Section 4.8). This is one of the reasons why we are interested in general D-rules.

The requirement, in the definitions of types L, S and D, that the literals in the head of the rule contain explainable predicate symbols is not an essential limitation. If, for instance, the predicate symbol in the head of $L \Leftarrow G$ is not explainable then this rule can be equivalently replaced by the C-rule $\bot \Leftarrow G \wedge \overline{L}$. If a rule has the form

$$L_1 \leftrightarrow L_2 \Leftarrow G$$

and the predicate symbol in L_1 is not explainable then the rule can be replaced by

$$\begin{array}{c} L_2 \Leftarrow G \wedge L_1, \\ \overline{L_2} \Leftarrow G \wedge \overline{L_1}. \end{array}$$

If a rule has the form

$$L_1 \vee \cdots \vee L_n \Leftarrow G$$

and the predicate symbol in L_1 is not explainable then the rule can be replaced by

$$L_2 \vee \cdots \vee L_n \Leftarrow G \wedge \overline{L_1}.$$

4.3 Translating C-Rules and L-Rules

The transformation described in this section generalizes McCain's translation, in the form described in Section 4.1.4, to first-order causal theories.

The operator Tr_c, which transforms any C-rule into a program rule, is defined by the formula

$$\mathrm{Tr}_c[\bot \Leftarrow G] = \leftarrow \neg G.$$

The operator Tr_l, which transforms any L-rule into a program rule, is defined by the formulas

$$\begin{aligned} \mathrm{Tr}_l[p(\mathbf{t}) \Leftarrow G] &= p(\mathbf{t}) \leftarrow \neg\neg G, \\ \mathrm{Tr}_l[\neg p(\mathbf{t}) \Leftarrow G] &= \widehat{p}(\mathbf{t}) \leftarrow \neg\neg G \end{aligned}$$

($\mathbf{t}$ is a tuple of terms).

If T is a causal theory consisting of C-rules and L-rules then its translation $\mathrm{Tr}[T]$ is the logic program obtained by conjoining

- the rules obtained by applying Tr_c to the C-rules of T,
- the rules obtained by applying Tr_l to the L-rules of T, and
- the completeness constraints

  $$\begin{aligned} &\bot \leftarrow p(\mathbf{x}) \land \widehat{p}(\mathbf{x}), \\ &\bot \leftarrow \neg p(\mathbf{x}) \land \neg\widehat{p}(\mathbf{x}) \end{aligned} \qquad (4.16)$$

  ($\mathbf{x}$ is a tuple of distinct object variables) for all explainable predicate symbols p of T.

Let $\mathbf{p}$ be the list of explainable predicate symbols p of T, and let $\widehat{\mathbf{p}}$ be the list of the corresponding predicate symbols $\widehat{p}$. Take the union of $\mathbf{p}$ and $\widehat{\mathbf{p}}$ to be the set of intensional predicates. Then the stable models of the logic program $\mathrm{Tr}[T]$ are "almost identical" to the models of T; the difference is due to the fact that the language of T does not contain the symbols $\widehat{\mathbf{p}}$. Let CC be the conjunction of all completeness constraints (4.16). Then the relationship between T and $\mathrm{Tr}[T]$ can be described as follows:

$$\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\mathrm{Tr}[T]] \textit{ is equivalent to } T \wedge CC. \tag{4.17}$$

This claim, expressing the soundness of our translation, is extended in Sections 4.4 and 4.5 to causal theories containing S-rules and D-rules.

Since the conjunction of the sentences corresponding to rules (4.16) is classically equivalent to

$$\forall \mathbf{x}(\widehat{p}(\mathbf{x}) \leftrightarrow \neg p(\mathbf{x})), \tag{4.18}$$

CC can be viewed as the conjunction of explicit definitions of the predicates $\widehat{\mathbf{p}}$ in terms of the predicates $\mathbf{p}$. Consequently the relationship (4.17) shows that $\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\mathrm{Tr}[T]]$ is a definitional extension of T. The models of $\mathrm{Tr}[T]$ that are stable relative to $\mathbf{p}\widehat{\mathbf{p}}$ can be characterized as the models of T extended by the interpretations of the predicates $\widehat{\mathbf{p}}$ that are provided by definitions (4.18).

Example 4.3.1 (Example 2.3.1, continued)**.** For causal theory T_0 where both p and q explainable, $\mathrm{Tr}[T_0]$ is the conjunction of formulas (4.15). The result of applying the operator $\mathrm{SM}_{pq\widehat{p}\widehat{q}}$ to this conjunction is equivalent to

$$p \wedge \neg q \wedge \neg\widehat{p} \wedge \widehat{q}.$$

Recall that T is equivalent to the first half of this conjunction (Section 2.3.1). The second half tells us that the truth values of $\widehat{p}$, $\widehat{q}$ are opposite to the truth values of p, q. In the only stable model of (4.15), p and $\widehat{q}$ are true, and $\widehat{p}$ and q

are false; if we "forget" the truth values of $\widehat{p}$ and $\widehat{q}$ then we will arrive at the model of T_0.

Example 4.3.2 (Example 2.3.2, continued)**.** Our translation turns the causal theory T_1 from Example 2.3.2 into the conjunction of the sentences

$$\begin{array}{l} \neg\neg\top \rightarrow p(a), \\ \forall x(\neg\neg\neg p(x) \rightarrow \widehat{p}(x)), \\ \forall x\neg(p(x) \wedge \widehat{p}(x)), \\ \forall x\neg(\neg p(x) \wedge \neg\widehat{p}(x)), \end{array}$$

or, after intuitionistically equivalent transformations,

$$\begin{array}{l} p(a), \\ \forall x(\neg p(x) \rightarrow \widehat{p}(x)), \\ \forall x\neg(p(x) \wedge \widehat{p}(x)), \\ \forall x\neg(\neg p(x) \wedge \neg\widehat{p}(x)). \end{array}$$

The result of applying $\mathrm{SM}_{p\widehat{p}}$ to the conjunction of these formulas is equivalent to the conjunction of (2.10) with the formula $\forall x(\widehat{p}(x) \leftrightarrow \neg p(x))$, which says that $\widehat{p}$ is the complement of p.

Example 4.3.3 (Example 2.3.5, continued)**.** Our translation turns causal theory (2.13) into the conjunction of the sentences

$$\begin{array}{l} \forall x(\neg\neg(toggle(x) \wedge \neg on_0(x)) \rightarrow on_1(x)), \\ \forall x(\neg\neg(toggle(x) \wedge on_0(x)) \rightarrow \widehat{on_1}(x)), \\ \forall x(\neg\neg(on_0(x) \wedge on_1(x)) \rightarrow on_1(x)), \\ \forall x(\neg\neg(\neg on_0(x) \wedge \neg on_1(x)) \rightarrow \widehat{on_1}(x)), \\ \forall x\neg(on_1(x) \wedge \widehat{on_1}(x)), \\ \forall x\neg(\neg on_1(x) \wedge \neg\widehat{on_1}(x)), \end{array} \tag{4.19}$$

or, equivalently,[2]

$$\begin{array}{l}\forall x(\mathit{toggle}(x) \land \neg on_0(x) \to on_1(x)),\\ \forall x(\mathit{toggle}(x) \land on_0(x) \to \widehat{on_1}(x)),\\ \forall x(on_0(x) \land \neg \widehat{on_1}(x) \to on_1(x)),\\ \forall x(\neg on_0(x) \land \neg on_1(x) \to \widehat{on_1}(x)),\\ \forall x \neg(on_1(x) \land \widehat{on_1}(x)),\\ \forall x \neg(\neg on_1(x) \land \neg \widehat{on_1}(x)).\end{array} \tag{4.20}$$

The result of applying $\mathrm{SM}_{on_1 \widehat{on_1}}$ to this program is equivalent to the conjunction of (2.14) with the formula $\forall x(\widehat{on_1}(x) \leftrightarrow \neg on_1(x))$, which says that $\widehat{on_1}$ is the complement of on_1.

Example 4.3.4 (Example 4.3.3, continued)**.** The constraint

$$\bot \Leftarrow \mathit{toggle}(\mathit{badswitch})$$

expresses that *badswitch* is stuck: the action of toggling it is not executable. If we add this constraint to the causal theory from Example 2.3.5 then the rule

$$\neg \mathit{toggle}(\mathit{badswitch})$$

will be added to its translation (4.20).

The bodies of causal rules in Examples 2.3.5 and 4.3.4 are syntactically simple: they are conjunctions of literals. The general definitions of a C-rule

[2]Removing the double negations in the first two lines of (4.19) is possible because neither *toggle* nor on_0 is intensional (see the comment on equivalent transformations of logic programs at the end of Section 2.4). In a similar way, the antecedent of the third implication in (4.19) can be replaced by $on_0(x) \land \neg\neg on_1(x)$; the equivalence between $\neg\neg on_1(x)$ and $\neg\widehat{on_1}(x)$ is intuitionistically entailed by the last two lines of (4.19). The fourth line of (4.19) is simplified in a similar way.

and an L-rule do not impose any restrictions on the form of the body, and in applications of causal logic to formalizing commonsense knowledge this generality is often essential. For instance, the statement "each position must have at least one neighbor" in the landscape structure of the Zoo World[3] would be represented in causal logic by a C-rule with a quantifier in the body.

4.4 Translating S-Rules

We will turn now to translating synonymy rules (Section 4.2). The operator Tr_s, transforming any such rule into a logic program, is defined by the formulas

$$\mathrm{Tr}_s[p_1(\mathbf{t}^1) \leftrightarrow p_2(\mathbf{t}^2) \Leftarrow G] = \mathrm{Tr}_s[\neg p_1(\mathbf{t}^1) \leftrightarrow \neg p_2(\mathbf{t}^2) \Leftarrow G]$$
$$= \begin{cases} p_2(\mathbf{t}^2) \leftarrow \neg\neg G \wedge p_1(\mathbf{t}^1) \\ p_1(\mathbf{t}^2) \leftarrow \neg\neg G \wedge p_2(\mathbf{t}^1) \\ \widehat{p_2}(\mathbf{t}^2) \leftarrow \neg\neg G \wedge \widehat{p_1}(\mathbf{t}^1) \\ \widehat{p_1}(\mathbf{t}^2) \leftarrow \neg\neg G \wedge \widehat{p_2}(\mathbf{t}^1) \end{cases}$$

$$\mathrm{Tr}_s[\neg p_1(\mathbf{t}^1) \leftrightarrow p_2(\mathbf{t}^2) \Leftarrow G] = \mathrm{Tr}_s[p_1(\mathbf{t}^1) \leftrightarrow \neg p_2(\mathbf{t}^2) \Leftarrow G]$$
$$= \begin{cases} p_2(\mathbf{t}^2) \leftarrow \neg\neg G \wedge \widehat{p_1}(\mathbf{t}^1) \\ p_1(\mathbf{t}^2) \leftarrow \neg\neg G \wedge \widehat{p_2}(\mathbf{t}^1) \\ \widehat{p_2}(\mathbf{t}^2) \leftarrow \neg\neg G \wedge p_1(\mathbf{t}^1) \\ \widehat{p_1}(\mathbf{t}^2) \leftarrow \neg\neg G \wedge p_2(\mathbf{t}^1) \end{cases}$$

($\mathbf{t}^1$, $\mathbf{t}^2$ are tuples of terms). The definition of program $\mathrm{Tr}[T]$ from Section 4.3 is extended to causal theories that may contain S-rules, besides C-rules and L-rules, by adding that $\mathrm{Tr}[T]$ includes also

- the rules obtained by applying Tr_s to the S-rules of T.

[3]The challenge of formalizing the Zoo World was proposed as part of the Logic Modelling Workshop (`http:/www/ida.liu.se/ext/etai/lmw/`). The possibility of addressing this challenge using CCALC is discussed in [1, Section 4].

Example 4.4.1 (Example 2.3.5, continued)**.** Extend the theory from Example 4.3.3 by the rule (2.15) where *dark* is explainable. The corresponding logic program is obtained from (4.20) by adding the rules

$$\begin{array}{l} on_1(myswitch) \leftarrow \widehat{dark}, \\ \widehat{dark} \leftarrow on_1(myswitch), \\ \widehat{on_1}(myswitch) \leftarrow dark, \\ dark \leftarrow \widehat{on_1}(myswitch), \\ \bot \leftarrow dark \wedge \widehat{dark}, \\ \bot \leftarrow \neg dark \wedge \neg\widehat{dark}. \end{array} \tag{4.21}$$

We will see that the soundness property (4.17) holds for arbitrary causal theories consisting of rules of types C, L, and S.

4.5 Translating D-Rules

A D-rule (Section 4.2) has the form

$$\bigvee_{A \in Pos} A \vee \bigvee_{A \in Neg} \neg A \Leftarrow G \tag{4.22}$$

for some sets *Pos*, *Neg* of atomic formulas.

If A is an atomic formula $p(\mathbf{t})$, where $p \in \mathbf{p}$ and $\mathbf{t}$ is a tuple of terms, then by $\widehat{A}$ we will denote the formula $\widehat{p}(\mathbf{t})$. The operator Tr_d transforms D-rule (4.22) into the program rule

$$\bigvee_{A \in Pos} A \vee \bigvee_{A \in Neg} \widehat{A} \leftarrow \neg\neg G \wedge \bigwedge_{A \in Pos} (\widehat{A} \vee \neg\widehat{A}) \wedge \bigwedge_{A \in Neg} (A \vee \neg A). \tag{4.23}$$

This is a generalization of the translation of propositional causal theories into logic programs due to [17].

Example 4.5.1. The result of applying Tr_d to the D-rule

$$p \vee \neg q \vee \neg r \Leftarrow s$$

is

$$p \vee \widehat{q} \vee \widehat{r} \leftarrow \neg\neg s \wedge (\widehat{p} \vee \neg\widehat{p}) \wedge (q \vee \neg q) \wedge (r \vee \neg r).$$

The number of "excluded middle formulas" conjoined with $\neg\neg G$ in (4.23) equals the number of disjunctive terms in the head of D-rule (4.22). In particular, if (4.22) is an L-rule then the antecedent of (4.23) contains one such formula. For instance, in application to the first rule of (2.9) Tr_d produces the program rule

$$p \leftarrow \neg\neg\neg q \wedge (\widehat{p} \vee \neg\widehat{p}),$$

which is more complex than the first rule of (4.15).

4.6 Soundness of Translation

For a fixed collection $\mathbf{p}$ of explainable symbols, let C, L, S, and D be finite sets of causal rules of types C, L, S, and D respectively. By $\mathrm{Tr}[C, L, S, D]$ we denote the logic program obtained by conjoining

- the rules obtained by applying Tr_c to all rules from C,
- the rules obtained by applying Tr_l to all rules from L,
- the programs obtained by applying Tr_s to all rules from S,
- the rules obtained by applying Tr_d to all rules from D,

- the completeness constraints (4.16) for all explainable symbols p.

Theorem 4.6.1 (Soundness of Translation). *For the causal theory T with the set of rules $C \cup L \cup S \cup D$,*

$$SM_{\mathbf{p}\widehat{\mathbf{p}}}[\mathrm{Tr}[C, L, S, D]] \text{ is equivalent to } T \wedge CC \tag{4.24}$$

In the special case when D is empty this theorem turns into the assertion stated at the end of Section 4.4.

4.7 Using Answer Set Solvers to Generate Models of a Causal Theory

The discussion of answer set solvers in this section, as almost any discussion of software, is somewhat informal. We assume here that the first-order language under consideration does not contain function constants of nonzero arity.

An answer set solver can be viewed as a system for generating stable models in the sense of Section 2.4, with three caveats. First, currently available solvers require that the input program have a syntactic form that is much more restrictive than the syntax of first-order logic.[4] Preprocessing based on intuitionistically equivalent transformations often helps us alleviate this difficulty. There exists a tool, called F2LP [42], that converts first-order formulas of a rather general kind into logic programs accepted by LPARSE.

[4]They also require that the input satisfy some safety conditions. See, for instance, Chapter 3 of the *SC*dlv manual, `http://www.dbai.tuwien.ac.at/proj/dlv/man/`.

The rules produced by the process described in the previous section have no existential quantifiers in their heads, and all quantifiers in their bodies are in the scope of negation. Consequently, these rules satisfy a syntactic condition that guarantees the correctness of the translation implemented in F2LP.

Second, answer set solvers represent stable models by sets of ground atoms. To introduce such a representation, we usually choose a finite set of object constants that includes all object constants occurring in the program, and restrict attention to Herbrand interpretations of the extended language. The `#domain` construct of LPARSE can be used to specify the object constants constituting the domain of the variables in the program.

Third, most existing answer set solvers are unaware of the possibility of non-intensional (or *extensional*) predicates. Treating a predicate constant as extensional can be simulated using a choice rule [20, Theorem 2]. There is also another approach to overcoming this limitation. Take a conjunction E of some ground atoms containing extensional predicates, and assume that we are interested in the Herbrand stable models of a program F that interpret the extensional predicates in accordance with E (every atom from E is true; all other atoms containing extensional predicates are false). Under some syntactic conditions,[5] these stable models are identical to the Herbrand stable models of $F \wedge E$ with all predicate constants treated as intensional. This can be proved using the splitting theorem (Fact 2.4.4).

[5]Specifically, under the assumption that every occurrence of every extensional predicate in F is in the scope of negation or in the antecedent of an implication.

```
u(a;b;c;d).
#domain u(X).
{q(X)} :- not p(X).
p(a;b).
```

Figure 4.1: Example 4.7.1 with a 4-element universe in the language of LPARSE

Example 4.7.1 (Example 2.4.2, continued)**.** We would like to find the stable models of (2.17), with q intensional, that have the universe $\{a, b, c, d\}$ and make p true on a, b and false on c, d. This is the same as to look for the Herbrand stable models of the formula

$$\forall x(\neg p(x) \to (q(x) \vee \neg q(x))) \wedge p(a) \wedge p(b),$$

with c and d viewed as object constants of the language along with a and b, and with both p and q taken to be intensional.

A representation of this example in the language of LPARSE is shown in Figure 4.1. The auxiliary predicate `u` describes the universe of the interpretations that we are interested in. The first line is shorthand for

```
u(a).  u(b).  u(c).  u(d).
```

and the last line is understood by LPARSE in a similar way.

Given this input, the answer set solver SMODELS generates 4 stable models, representing the subsets of $\{a, b, c, d\}$ that are disjoint from $\{a, b\}$:

```
Answer: 1
```

```
Stable Model: p(b) p(a) u(d) u(c) u(b) u(a)
Answer: 2
Stable Model: p(b) p(a) q(d) u(d) u(c) u(b) u(a)
Answer: 3
Stable Model: p(b) p(a) q(c) u(d) u(c) u(b) u(a)
Answer: 4
Stable Model: p(b) p(a) q(d) q(c) u(d) u(c) u(b) u(a)
```

In application to the logic program obtained from a causal theory T, this process often allows us to find the models of T with a given universe and given extents of extensional predicates.

Example 4.7.2 (Example 4.4.1, continued)**.** There are two switches, *myswitch* and *hisswitch*. It is dark in my room at time 1 if and only if *myswitch* is not *on* at time 1. At time 0, both switches are *on*; then *hisswitch* is toggled, and *myswitch* is not. Is it dark in my room at time 1? We would like to answer this question using answer set programming.

This example of commonsense reasoning involves inertia (the value of the fluent $on(myswitch)$ does not change because this fluent is not affected by the action that is executed) and indirect effects of actions: whether or not it is dark in the room at time 1 after performing some actions is determined by the effect of these actions on the fluent $on(myswitch)$.

Mathematically, we are talking here about the causal theory T with rules (2.13) and (2.15), with the object constant *hisswitch* added to the language, and with the explainable symbols on_1 and $dark$. We are interested in the Herbrand models of T in which the extents of the extensional predicates are described by the atoms

$$on_0(myswitch),\ on_0(hisswitch),\ toggle(hisswitch).$$

As we have seen, the logic program Tr[T] is equivalent to the conjunction of rules (4.20) and (4.21). The corresponding LPARSE input file is shown in Figure 4.2. In this file, the “true negation” symbol `-` is used in the ASCII representations of the symbols $\widehat{on_1}$ and $\widehat{dark}$; the LPARSE counterparts of the rules

$$\begin{array}{l}\bot \leftarrow on_1(x) \wedge \widehat{on_1}(x),\\ \bot \leftarrow dark \wedge \widehat{dark}\end{array}$$

are dropped, because such “coherence” conditions are verified by the system automatically.

Given this input, SMODELS generates the only model of T satisfying the given conditions:

```
Answer: 1
Stable Model: -on1(hisswitch) on1(myswitch) -dark
toggle(hisswitch) on0(hisswitch) on0(myswitch)
u(hisswitch) u(myswitch)
```

The presence of `-dark` in this model tells us that it is not dark in the room at time 1.

```
u(myswitch;hisswitch).
#domain u(X).

on1(X) :- toggle(X), not on0(X).
-on1(X) :- toggle(X), on0(X).
on1(X) :- on0(X), not -on1(X).
-on1(X) :- not on0(X), not on1(X).
:- not on1(X), not -on1(X).

on1(myswitch) :- -dark.
-dark :- on1(myswitch).
-on1(myswitch) :- dark.
dark :- -on1(myswitch).
:- not dark, not -dark.

on0(myswitch;hisswitch).
toggle(hisswitch).
```

Figure 4.2: Example 4.7.2 with two switches in the language of LPARSE

The example above is an example of "one-step temporal projection"—predicting the value of a fluent after performing a single action in a given state. Some other kinds of temporal reasoning and planning can be performed by generating models of simple modifications of the given causal theory [33, Section 3.3]; this is one of the ideas behind the design of CCALC and COALA. McCain's translation reviewed in Section 4.1 and its generalization presented above allow us to solve such problems automatically using an answer set solver.

4.8 Proof of Theorem 4.6.1

To prove claim (4.24), which expresses the soundness of our translation, we will first establish it for the case when $C = L = S = \emptyset$ (Section 4.8.1). In this "leading special case" all rules of the given causal theory are D-rules, and they are converted to program rules using the translation Tr_d. Then we will derive the soundness theorem in full generality (Section 4.8.2).

4.8.1 Leading Special Case

Let T be a finite set of causal rules of the form (4.22). Let Π be the conjunction of the corresponding program rules (4.23), and let CC, as before, stand for the conjunction of the completeness constraints (4.16) for all explainable symbols p of T. We want to show that

$$\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\Pi \wedge CC] \textit{ is equivalent to } T \wedge CC. \tag{4.25}$$

The key steps in the proof below are Lemma 4.8.5 (one half of the equivalence) and Lemma 4.8.8 (the other half).

In the statement of the following lemma, $\neg\mathbf{p}$ stands for the list of predicate expressions[6] $\lambda\mathbf{x}\neg p(\mathbf{x})$, where $\mathbf{x}$ is a list of distinct object variables, for all p from $\mathbf{p}$. By $\upsilon\mathbf{p}$, $\upsilon\widehat{\mathbf{p}}$ we denote the lists of predicate variables used in the second-order formula $\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\Pi \wedge CC]$ (see Section 2.4).

[6]See [43, Section 3.1].

Lemma 4.8.1. *Formula* $(\upsilon \boldsymbol{p}, \upsilon \widehat{\mathbf{p}}) < (\boldsymbol{p}, \neg \boldsymbol{p})$ *is equivalent to*

$$\bigvee_{p \in \boldsymbol{p}} (((\upsilon \boldsymbol{p}, \upsilon \widehat{\boldsymbol{p}}) \leq (\boldsymbol{p}, \neg \boldsymbol{p})) \wedge \exists \boldsymbol{x}(\neg \upsilon p(\boldsymbol{x}) \wedge \neg \upsilon \widehat{p}(\boldsymbol{x}))).$$

Proof. Note first that

$$\begin{aligned}
&(\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) < (\mathbf{p}, \neg \mathbf{p}) \\
&\Leftrightarrow ((\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) \leq (\mathbf{p}, \neg \mathbf{p})) \wedge \neg\, ((\mathbf{p}, \neg \mathbf{p}) \leq (\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}})) \\
&\Leftrightarrow ((\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) \leq (\mathbf{p}, \neg \mathbf{p})) \wedge \bigvee\nolimits_{p \in \mathbf{p}} \exists \mathbf{x}((p(\mathbf{x}) \wedge \neg \upsilon p(\mathbf{x})) \vee (\neg p(\mathbf{x}) \wedge \neg \upsilon \widehat{p}(\mathbf{x}))) \\
&\Leftrightarrow \bigvee\nolimits_{p \in \mathbf{p}} (((\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) \leq (\mathbf{p}, \neg \mathbf{p})) \wedge \exists \mathbf{x}((p(\mathbf{x}) \wedge \neg \upsilon p(\mathbf{x})) \vee (\neg p(\mathbf{x}) \wedge \neg \upsilon \widehat{p}(\mathbf{x})))).
\end{aligned}$$

The disjunction after $\exists \mathbf{x}$ is equivalent to

$$(p(\mathbf{x}) \vee \neg \upsilon \widehat{p}(\mathbf{x})) \wedge (\neg \upsilon p(\mathbf{x}) \vee \neg p(\mathbf{x})) \wedge (\neg \upsilon p(\mathbf{x}) \vee \neg \upsilon \widehat{p}(\mathbf{x})). \tag{4.26}$$

Since $(\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) \leq (\mathbf{p}, \neg \mathbf{p})$ entails

$$\upsilon p(\mathbf{x}) \rightarrow p(\mathbf{x}) \text{ and } \upsilon \widehat{p}(\mathbf{x}) \rightarrow \neg p(\mathbf{x}),$$

the first conjunctive term of (4.26) can be rewritten as $\neg \upsilon \widehat{p}(\mathbf{x})$, and the second term as $\neg \upsilon p(\mathbf{x})$, so that (4.26) will turn into $\neg \upsilon p(\mathbf{x}) \wedge \neg \upsilon \widehat{p}(\mathbf{x})$.

For any formula F, by $F_{\Sigma 1}$ we denote the formula

$$F^{(\upsilon \mathbf{p})(\upsilon \widehat{\mathbf{p}})}_{(\upsilon \mathbf{p} \wedge \mathbf{p})(\neg \upsilon \mathbf{p} \wedge \neg \mathbf{p})}$$

where $\upsilon \mathbf{p} \wedge \mathbf{p}$ is understood as the list of predicate expressions

$$\lambda \mathbf{x}(\upsilon p(\mathbf{x}) \wedge p(\mathbf{x}))$$

for all $p \in \mathbf{p}$, and $\neg \upsilon \mathbf{p} \wedge \neg \mathbf{p}$ is understood in a similar way.[7]

[7]For the definition of $F^{\mathbf{p}}_{\upsilon \mathbf{p}}$ see Section 2.3.1.

Lemma 4.8.2. *Formula*

$$((\upsilon \boldsymbol{p}, \upsilon \widehat{\mathbf{p}}) < (\boldsymbol{p}, \neg \boldsymbol{p}))_{\Sigma 1}$$

is equivalent to $\upsilon \boldsymbol{p} \neq \boldsymbol{p}$.

Proof. In view of Lemma 4.8.1, $((\upsilon\mathbf{p}, \upsilon\widehat{\mathbf{p}}) < (\mathbf{p}, \neg\mathbf{p}))_{\Sigma 1}$ is equivalent to the disjunction of the formulas

$$\begin{array}{ll}\left(\bigwedge_{p\in\mathbf{p}} \forall\mathbf{x}(\upsilon p(\mathbf{x}) \to p(\mathbf{x}))_{\Sigma 1}\right) & \wedge \quad \left(\bigwedge_{p\in\mathbf{p}} \forall\mathbf{x}(\upsilon\widehat{p}(\mathbf{x}) \to \neg p(\mathbf{x}))_{\Sigma 1}\right) \\ & \wedge \quad \exists\mathbf{x}(\neg\upsilon p(\mathbf{x}) \wedge \neg\upsilon\widehat{p}(\mathbf{x}))_{\Sigma 1}\end{array} \tag{4.27}$$

for all $p \in \mathbf{p}$. It is easy to verify that

$$\begin{array}{rcl}(\upsilon p(\mathbf{x}) \to p(\mathbf{x}))_{\Sigma 1} & = & (\upsilon p(\mathbf{x}) \wedge p(\mathbf{x}) \to p(\mathbf{x})) \quad \Leftrightarrow \quad \top\,, \\ (\upsilon\widehat{p}(\mathbf{x}) \to \neg p(\mathbf{x}))_{\Sigma 1} & = & (\neg\upsilon p(\mathbf{x}) \wedge \neg p(\mathbf{x}) \to \neg p(\mathbf{x})) \quad \Leftrightarrow \quad \top, \\ (\neg\upsilon p(\mathbf{x}) \wedge \neg\upsilon\widehat{p}(\mathbf{x}))_{\Sigma 1} & \Leftrightarrow & ((\neg\upsilon p(\mathbf{x}) \vee \neg p(\mathbf{x})) \wedge \neg(\neg\upsilon p(\mathbf{x}) \wedge \neg p(\mathbf{x}))) \\ & \Leftrightarrow & (\upsilon p(\mathbf{x}) \leftrightarrow \neg p(\mathbf{x})) \\ & \Leftrightarrow & \neg(\upsilon p(\mathbf{x}) \leftrightarrow p(\mathbf{x})).\end{array}$$

Therefore (4.27) is equivalent to $\exists\mathbf{x}\neg(\upsilon p(\mathbf{x}) \leftrightarrow p(\mathbf{x}))$, so that the disjunction of all formulas (4.27) is equivalent to $\upsilon\mathbf{p} \neq \mathbf{p}$.

If A is an atomic formula $p(\mathbf{t})$, where $p \in \mathbf{p}$ and $\mathbf{t}$ is a tuple of terms, then we will write υA for $\upsilon p(\mathbf{t})$, and $\widehat{A}$ for $\upsilon\widehat{p}(\mathbf{t})$. By $\widetilde{\forall}_{obj} F$ we denote the formula $\forall\mathbf{x}F$, where $\mathbf{x}$ is list of all free object variables of F ("object-level universal closure").

Define $H(\upsilon\mathbf{p}, \upsilon\widehat{\mathbf{p}})$ to be the conjunction of the implications

$$\widetilde{\forall}_{obj}\left(G \to \bigvee_{A\in Pos} ((\upsilon\widehat{A} \vee A) \to \upsilon A) \vee \bigvee_{A\in Neg} ((\upsilon A \vee \neg A) \to \upsilon\widehat{A})\right) \tag{4.28}$$

for all rules (4.22) in T.

Lemma 4.8.3. *Formula* $SM_{\boldsymbol{p}\widehat{\mathbf{p}}}[\Pi \wedge CC]$ *is equivalent to*

$$\Pi \wedge CC \wedge \forall(\upsilon \boldsymbol{p})(\upsilon \widehat{\mathbf{p}})(((\upsilon \boldsymbol{p}, \upsilon \widehat{\mathbf{p}}) < (\boldsymbol{p}, \neg \boldsymbol{p})) \rightarrow \neg H(\upsilon \boldsymbol{p}, \upsilon \widehat{\mathbf{p}})). \tag{4.29}$$

Proof. Every occurrence of every intensional predicate in CC is in the scope of a negation. Consequently $\text{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\Pi \wedge CC]$ is

$$\Pi \wedge CC \wedge \neg\exists(\upsilon \mathbf{p})(\upsilon \widehat{\mathbf{p}})(((\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) < (\mathbf{p}, \widehat{\mathbf{p}})) \wedge \Pi^\diamond(\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) \wedge CC),$$

which is equivalent to

$$\Pi \wedge CC \wedge \forall(\upsilon \mathbf{p})(\upsilon \widehat{\mathbf{p}})(((\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) < (\mathbf{p}, \neg \mathbf{p})) \rightarrow \neg\Pi^\diamond(\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}})).$$

We will conclude the proof by showing that CC entails

$$\Pi^\diamond(\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}) \leftrightarrow H(\upsilon \mathbf{p}, \upsilon \widehat{\mathbf{p}}).$$

The left-hand side of this equivalence is the conjunction of the formulas

$$\widetilde{\forall}_{obj}\left(\neg\neg G \wedge \bigwedge_{A \in Pos} (\upsilon \widehat{A} \vee \neg \widehat{A}) \wedge \bigwedge_{A \in Neg} (\upsilon A \vee \neg A) \rightarrow \bigvee_{A \in Pos} \upsilon A \vee \bigvee_{A \in Neg} \upsilon \widehat{A}\right)$$

for all rules (4.22) in T. Under the assumption CC this formula can be rewritten as

$$\widetilde{\forall}_{obj}\left(G \rightarrow \bigvee_{A \in Pos} \neg(\upsilon \widehat{A} \vee A) \vee \bigvee_{A \in Neg} \neg(\upsilon A \vee \neg A) \vee \bigvee_{A \in Pos} \upsilon A \vee \bigvee_{A \in Neg} \upsilon \widehat{A}\right).$$

The last formula is equivalent to

$$\widetilde{\forall}_{obj}\left(G \rightarrow \bigvee_{A \in Pos} (\neg(\upsilon \widehat{A} \vee A) \vee \upsilon A) \vee \bigvee_{A \in Neg} (\neg(\upsilon A \vee \neg A) \vee \upsilon \widehat{A})\right).$$

and consequently to (4.28).

Lemma 4.8.4. *$T^\dagger(\upsilon\boldsymbol{p})$ is equivalent to $H(\upsilon\boldsymbol{p}, \upsilon\widehat{\mathbf{p}})_{\Sigma 1}$.*

Proof. Formula $T^\dagger(\upsilon\mathbf{p})$ is the conjunction of the formulas

$$\widetilde{\forall}_{obj}\left(G \to \bigvee_{A\in Pos} \upsilon A \vee \bigvee_{A\in Neg} \neg\upsilon A\right) \tag{4.30}$$

for all rules (4.22) in T. On the other hand, $H(\upsilon\mathbf{p}, \upsilon\widehat{\mathbf{p}})_{\Sigma 1}$ is the conjunction of the formulas

$$\widetilde{\forall}_{obj}\left(G \to \bigvee_{A\in Pos} ((\upsilon\widehat{A} \vee A) \to \upsilon A)_{\Sigma 1} \vee \bigvee_{A\in Neg} ((\upsilon A \vee \neg A) \to \upsilon\widehat{A})_{\Sigma 1}\right) \tag{4.31}$$

for all rules (4.22) in T. It remains to observe that

$$\begin{array}{rcl} ((\upsilon\widehat{A} \vee A) \to \upsilon A)_{\Sigma 1} & = & (\neg\upsilon A \wedge \neg A) \vee A \to \upsilon A \wedge A \\ & \Leftrightarrow & \neg\upsilon A \vee A \to \upsilon A \wedge A \\ & \Leftrightarrow & (\upsilon A \wedge \neg A) \vee (\upsilon A \wedge A) \\ & \Leftrightarrow & \upsilon A, \end{array}$$

and that, similarly, $((\upsilon A \vee \neg A) \to \upsilon\widehat{A})_{\Sigma 1}$ is equivalent to $\neg\upsilon A$.

Lemma 4.8.5. *$SM_{\boldsymbol{p}\widehat{\mathbf{p}}}[\Pi \wedge CC] \models T \wedge CC$.*

Proof. Recall that, according to Lemma 4.8.3, $\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\Pi \wedge CC]$ is equivalent to (4.29). The second conjunctive term of (4.29) is CC. The first conjunctive term is equivalent to $T^\dagger(\mathbf{p})$. From the other two terms we conclude:

$$\forall\upsilon\mathbf{p}(((\upsilon\mathbf{p}, \upsilon\widehat{\mathbf{p}}) < (\mathbf{p}, \widehat{\mathbf{p}}))_{\Sigma 1} \to \neg H(\upsilon\mathbf{p}, \upsilon\widehat{\mathbf{p}})_{\Sigma 1}).$$

By Lemma 4.8.2 and Lemma 4.8.4, this formula is equivalent to

$$\forall\upsilon\mathbf{p}((\upsilon\mathbf{p} \neq \mathbf{p}) \to \neg T^\dagger(\upsilon\mathbf{p})),$$

and consequently to

$$\forall \upsilon\mathbf{p}(T^{\dagger}(\upsilon\mathbf{p}) \rightarrow (\upsilon\mathbf{p} = \mathbf{p})).$$

The conjunction of the last formula with $T^{\dagger}(\mathbf{p})$ is equivalent to (2.3).

For any formula F, by $F_{\Sigma 2}$ we denote the formula

$$F^{\upsilon\mathbf{p}}_{(((\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}})\leq(\mathbf{p},\neg\mathbf{p}))\wedge\neg\upsilon\mathbf{p}\wedge\neg\upsilon\widehat{\mathbf{p}})\leftrightarrow\neg\mathbf{p}}$$

where the subscript

$$(((\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}}) \leq (\mathbf{p},\neg\mathbf{p})) \wedge \neg\upsilon\mathbf{p} \wedge \neg\upsilon\widehat{\mathbf{p}}) \leftrightarrow \neg\mathbf{p}$$

is understood as the list of predicate expressions

$$\lambda\mathbf{x}((((\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}}) \leq (\mathbf{p},\neg\mathbf{p})) \wedge \neg\upsilon p(\mathbf{x}) \wedge \neg\upsilon\widehat{p}(\mathbf{x})) \leftrightarrow \neg p(\mathbf{x}))$$

for all $p \in \mathbf{p}$.

Lemma 4.8.6. *Formula*

$$(\upsilon\boldsymbol{p} \neq \boldsymbol{p})_{\Sigma 2}$$

is equivalent to $(\upsilon\boldsymbol{p},\upsilon\widehat{\mathbf{p}}) < (\boldsymbol{p},\neg\boldsymbol{p})$.

Proof. Formula $(\upsilon\mathbf{p} \neq \mathbf{p})_{\Sigma 2}$ is equivalent to

$$\bigvee_{p\in\mathbf{p}} \exists\mathbf{x}(\upsilon p(\mathbf{x}) \leftrightarrow \neg p(\mathbf{x}))_{\Sigma 2}$$

that is,

$$\bigvee_{p\in\mathbf{p}} \exists\mathbf{x}((((\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}}) \leq (\mathbf{p},\neg\mathbf{p})) \wedge \neg\upsilon p(\mathbf{x}) \wedge \neg\upsilon\widehat{p}(\mathbf{x}) \leftrightarrow \neg p(\mathbf{x})) \leftrightarrow \neg p(\mathbf{x})).$$

This formula can be equivalently rewritten as

$$\bigvee_{p\in\mathbf{p}}(((\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}})\leq(\mathbf{p},\neg\mathbf{p}))\wedge\exists\mathbf{x}(\neg\upsilon p(\mathbf{x})\wedge\neg\upsilon\widehat{p}(\mathbf{x}))),$$

which is equivalent to $(\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}})<(\mathbf{p},\neg\mathbf{p})$ by Lemma 4.8.1.

Lemma 4.8.7. *The implication*

$$(\upsilon\boldsymbol{p},\upsilon\widehat{\mathbf{p}})\leq(\boldsymbol{p},\neg\boldsymbol{p})\rightarrow(T^{\dagger}(\upsilon\boldsymbol{p})_{\Sigma2}\leftrightarrow H(\upsilon\boldsymbol{p},\upsilon\widehat{\mathbf{p}}))$$

is logically valid.

Proof. Recall that $T^{\dagger}(\upsilon\mathbf{p})$ is the conjunction of implications (4.30) for all rules (4.22) in T. Consequently $T^{\dagger}(\upsilon\mathbf{p})_{\Sigma2}$ is the conjunction of the formulas

$$\widetilde{\forall}_{obj}\left(G\rightarrow\bigvee_{A\in Pos}(\upsilon A)_{\Sigma2}\vee\bigvee_{A\in Neg}\neg(\upsilon A)_{\Sigma2}\right),$$

that is to say,

$$\begin{aligned}\widetilde{\forall}_{obj}(G\rightarrow\ &\textstyle\bigvee_{A\in Pos}((((\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}})\leq(\mathbf{p},\neg\mathbf{p}))\wedge\neg\upsilon A\wedge\neg\upsilon\widehat{A})\leftrightarrow\neg A)\vee\\ &\textstyle\bigvee_{A\in Neg}\neg((((\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}})\leq(\mathbf{p},\neg\mathbf{p}))\wedge\neg\upsilon A\wedge\neg\upsilon\widehat{A})\leftrightarrow\neg A).\end{aligned}$$

Under the assumption

$$(\upsilon\mathbf{p},\upsilon\widehat{\mathbf{p}})\leq(\mathbf{p},\neg\mathbf{p})\tag{4.32}$$

the last formula can be equivalently rewritten as

$$\widetilde{\forall}_{obj}\left(G\rightarrow\bigvee_{A\in Pos}((\upsilon A\vee\upsilon\widehat{A})\leftrightarrow A)\vee\bigvee_{A\in Neg}((\upsilon A\vee\upsilon\widehat{A})\leftrightarrow\neg A)\right).$$

It remains to check that, under assumption (4.32),

$$(\upsilon A\vee\upsilon\widehat{A})\leftrightarrow A\tag{4.33}$$

can be equivalently rewritten as

$$\upsilon\widehat{A} \vee A \rightarrow \upsilon A, \tag{4.34}$$

and

$$\upsilon A \vee \upsilon\widehat{A} \leftrightarrow \neg A \tag{4.35}$$

can be rewritten as

$$\upsilon A \vee \neg A \rightarrow \upsilon\widehat{A}. \tag{4.36}$$

Formula (4.33) is equivalent to

$$(\upsilon A \rightarrow A) \wedge (\upsilon\widehat{A} \rightarrow A) \wedge (A \rightarrow \upsilon A \vee \upsilon\widehat{A}). \tag{4.37}$$

Since assumption (4.32) entails $\upsilon A \rightarrow A$ and $\upsilon\widehat{A} \rightarrow \neg A$, formula (4.37) can be rewritten as

$$\neg\upsilon\widehat{A} \wedge (A \rightarrow \upsilon A). \tag{4.38}$$

On the other hand, formula (4.34) is equivalent to

$$(\upsilon\widehat{A} \rightarrow \upsilon A) \wedge (A \rightarrow \upsilon A),$$

which, under assumption (4.32), can be rewritten as (4.38) as well. In a similar way, each of the formulas (4.35), (4.36) can be transformed into

$$\neg\upsilon A \wedge (\neg A \rightarrow \upsilon\widehat{A}).$$

Lemma 4.8.8. $T \wedge CC \models SM_{\mathbf{p}\widehat{\mathbf{p}}}[\Pi \wedge CC]$.

Proof. Recall that T is equivalent to

$$T^\dagger(\mathbf{p}) \wedge \forall \upsilon\mathbf{p}(T^\dagger(\upsilon\mathbf{p}) \to (\upsilon\mathbf{p} = \mathbf{p})). \tag{4.39}$$

Since the first conjunctive term is equivalent to Π, $T \wedge CC$ entails

$$\Pi \wedge CC. \tag{4.40}$$

From the second conjunctive term of (4.39) we conclude

$$T^\dagger(\upsilon\mathbf{p})_{\Sigma 2} \to (\upsilon\mathbf{p} = \mathbf{p})_{\Sigma 2}$$

and consequently

$$\forall(\upsilon\mathbf{p})(\upsilon\widehat{\mathbf{p}})((\upsilon\mathbf{p} \neq \mathbf{p})_{\Sigma 2} \to \neg T^\dagger(\upsilon\mathbf{p})_{\Sigma 2}).$$

By Lemma 4.8.6, this is equivalent to

$$\forall(\upsilon\mathbf{p})(\upsilon\widehat{\mathbf{p}})(((\upsilon\mathbf{p}, \upsilon\widehat{\mathbf{p}}) < (\mathbf{p}, \neg\mathbf{p})) \to \neg T^\dagger(\upsilon\mathbf{p})_{\Sigma 2})$$

and, by Lemma 4.8.7, to

$$\forall(\upsilon\mathbf{p})(\upsilon\widehat{\mathbf{p}})(((\upsilon\mathbf{p}, \upsilon\widehat{\mathbf{p}}) < (\mathbf{p}, \neg\mathbf{p})) \to \neg H(\upsilon\mathbf{p}, \upsilon\widehat{\mathbf{p}})).$$

By Lemma 4.8.3, the conjunction of this formula with (4.40) is equivalent to $\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\Pi \wedge CC]$.

Assertion (4.25) follows from Lemmas 4.8.5 and 4.8.8.

4.8.2 General Case

Lemma 4.8.9. *For any C-rule R, $\mathrm{Tr}_c[R]$ is intuitionistically equivalent to $\mathrm{Tr}_d[R]$.*

Proof. If R is $\bot \Leftarrow G$ then $\mathrm{Tr}_c[R]$ is $\widetilde{\forall}\neg G$, and $\mathrm{Tr}_d[R]$ is $\widetilde{\forall}(\neg\neg G \rightarrow \bot)$.

Lemma 4.8.10. *For any L-rule R, the conjunction CC of completeness constraints intuitionistically entails*

$$\mathrm{Tr}_l[R] \leftrightarrow \mathrm{Tr}_d[R].$$

Proof. If R is $p(\mathbf{t}) \Leftarrow G$ then $\mathrm{Tr}_l[R]$ is

$$\widetilde{\forall}(\neg\neg G \rightarrow p(\mathbf{t})),$$

and $\mathrm{Tr}_d[R]$ is

$$\widetilde{\forall}(\neg\neg G \wedge (\widehat{p}(\mathbf{t}) \vee \neg\widehat{p}(\mathbf{t})) \rightarrow p(\mathbf{t})).$$

Since CC intuitionistically entails

$$\neg(p(\mathbf{t}) \leftrightarrow \widehat{p}(\mathbf{t})), \tag{4.41}$$

it is sufficient to check that $p(\mathbf{t})$ can be derived from (4.41) and

$$\widehat{p}(\mathbf{t}) \vee \neg\widehat{p}(\mathbf{t}) \rightarrow p(\mathbf{t}) \tag{4.42}$$

by the deductive means of intuitionistic propositional logic. Since (4.42) is equivalent to $p(\mathbf{t})$ in classical propositional logic, it is easy to see that $\neg\widehat{p}(\mathbf{t})$ can be derived from (4.41) and (4.42) in classical propositional logic. By

Glivenko's theorem,[8] it follows that it can be derived intuitionistically as well. Since $p(\mathbf{t})$ is intuitionistically derivable from (4.42) and $\neg\widehat{p}(\mathbf{t})$, we can conclude that $p(\mathbf{t})$ is intuitionistically derivable from (4.41) and (4.42).

The case when R is $\neg p(\mathbf{t}) \Leftarrow G$ is similar.

Lemma 4.8.11. *If R is an S-rule*

$$L_1 \leftrightarrow L_2 \Leftarrow G \tag{4.43}$$

and R_1, R_2 are the D-rules

$$L_1 \vee \overline{L_2} \Leftarrow G \quad \textit{and} \quad \overline{L_1} \vee L_2 \Leftarrow G \tag{4.44}$$

then the conjunction CC of completeness constraints intuitionistically entails

$$\mathrm{Tr}_s[R] \leftrightarrow \mathrm{Tr}_d[R_1] \wedge \mathrm{Tr}_d[R_2].$$

Proof. If each of the literals L_i is an atom A_i then $\mathrm{Tr}_s[R]$ is the conjunction of the formulas

$$\begin{array}{l}\widetilde{\forall}(\neg\neg G \wedge A_1 \rightarrow A_2),\\ \widetilde{\forall}(\neg\neg G \wedge A_2 \rightarrow A_1),\\ \widetilde{\forall}(\neg\neg G \wedge \widehat{A_1} \rightarrow \widehat{A_2}),\\ \widetilde{\forall}(\neg\neg G \wedge \widehat{A_2} \rightarrow \widehat{A_1}),\end{array} \tag{4.45}$$

$\mathrm{Tr}_d[R_1]$ is

$$\widetilde{\forall}(\neg\neg G \wedge (\widehat{A_1} \vee \neg\widehat{A_1}) \wedge (A_2 \vee \neg A_2) \rightarrow A_1 \vee \widehat{A_2}), \tag{4.46}$$

[8]This theorem [35], [63, Theorem 3.1] asserts that if a formula beginning with negation can be derived from a set Γ of formulas in classical propositional logic then it can be derived from Γ in intuitionistic propositional logic as well.

and $\mathrm{Tr}_d[R_2]$ is

$$\widetilde{\forall}(\neg\neg G \wedge (A_1 \vee \neg A_1) \wedge (\widehat{A_2} \vee \neg\widehat{A_2}) \rightarrow \widehat{A_1} \vee A_2). \tag{4.47}$$

We need to show that CC intuitionistically entails the equivalence between the conjunction of formulas (4.45) and the conjunction of formulas (4.46), (4.47). Since CC intuitionistically entails

$$\neg(A_1 \leftrightarrow \widehat{A_1}) \tag{4.48}$$

and

$$\neg(A_2 \leftrightarrow \widehat{A_2}), \tag{4.49}$$

it is sufficient to check that the conjunction of formulas (4.48), (4.49),

$$A_1 \leftrightarrow A_2 \tag{4.50}$$

and

$$\widehat{A_1} \leftrightarrow \widehat{A_2} \tag{4.51}$$

is equivalent in intuitionistic propositional logic to the conjunction of formulas (4.48), (4.49),

$$(\widehat{A_1} \vee \neg\widehat{A_1}) \wedge (A_2 \vee \neg A_2) \rightarrow A_1 \vee \widehat{A_2} \tag{4.52}$$

and

$$(A_1 \vee \neg A_1) \wedge (\widehat{A_2} \vee \neg\widehat{A_2}) \rightarrow \widehat{A_1} \vee A_2. \tag{4.53}$$

Left-to-right: Assume (4.48)–(4.51) and

$$(\widehat{A_1} \vee \neg\widehat{A_1}) \wedge (A_2 \vee \neg A_2); \tag{4.54}$$

our goal is to derive intuitionistically $A_1 \vee \widehat{A_2}$. Consider two cases, in accordance with the first disjunction in (4.54). *Case 1:* $\widehat{A_1}$. Then, by (4.51), $\widehat{A_2}$, and consequently $A_1 \vee \widehat{A_2}$. *Case 2:* $\neg\widehat{A_1}$. Consider two cases, in accordance with the second disjunction in (4.54). *Case 2.1:* A_2. Then, by (4.50), A_1, and consequently $A_1 \vee \widehat{A_2}$. *Case 2.2:* $\neg A_2$. Then, by (4.50), $\neg A_1$, which contradicts (4.48).

Thus we proved that (4.52) is intuitionistically derivable from (4.48)–(4.51). The proof for (4.53) is similar.

Right-to-left: Let Γ be the set consisting of formulas (4.48), (4.49), (4.52), (4.53) and A_1. We claim that A_2 can be derived from Γ in intuitionistic propositional logic. Note that, classically,

- Formula (4.48) is equivalent to $A_1 \leftrightarrow \neg\widehat{A_1}$,
- Formula (4.49) is equivalent to $A_2 \leftrightarrow \neg\widehat{A_2}$, and
- Formula (4.53) is equivalent to $\widehat{A_1} \vee A_2$.

It follows that $\neg\widehat{A_2}$ is derivable from Γ in classical propositional logic. By Glivenko's theorem, it follows that $\neg\widehat{A_2}$ is derivable from Γ intuitionistically as well. Hence the antecedent of (4.53) is an intuitionistic consequence of Γ, and so is the consequent $\widehat{A_1} \vee A_2$. In combination with A_1 and (4.48), this gives us A_2.

We conclude that $A_1 \rightarrow A_2$ is intuitionsistically derivable from (4.48), (4.49), (4.52) and (4.53). The derivability of the implication $A_2 \rightarrow A_1$ from these formulas can be proved in a similar way. Thus (4.50) is an intuitionistic consequence of (4.48), (4.49), (4.52), and (4.53).

The derivability of (4.51) from these formulas in propositional intuitionistic logic is proved in a similar way.

The cases when the literals L_i are negative, or when one of them is positive and the other is negative, are similar.

Proof of the soundness property (4.24). Let C, L, S, and D be sets of causal rules of types C, L, S, and D respectively, and let T be the causal theory with the set of rules $C \cup L \cup S \cup D$. Consider the causal theory T' obtained from T by replacing each rule (4.43) from S with the corresponding rules (4.44). According to the result (4.25) of Section 4.8.1,

$$\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\Pi \wedge CC] \text{ is equivalent to } T' \wedge CC,$$

where Π is the conjunction of the program rules $\mathrm{Tr}_d[R]$ for all rules R of T'. It is clear that $\Pi \wedge CC$ is $\mathrm{Tr}[T']$, and that T' is equivalent to T. Consequently

$$\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\mathrm{Tr}[T']] \text{ is equivalent to } T \wedge CC. \tag{4.55}$$

On the other hand, Lemmas 4.8.9, 4.8.10 and 4.8.11 show that the formulas $\mathrm{Tr}[T']$ and $\mathrm{Tr}[C, L, S, D]$ are intuitionistically equivalent to each other, because each of them contains CC as a conjunctive term. It follows that

$$\mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\mathrm{Tr}[T']] \text{ is equivalent to } \mathrm{SM}_{\mathbf{p}\widehat{\mathbf{p}}}[\mathrm{Tr}[C, L, S, D]]. \tag{4.56}$$

Assertion (4.24) follows from (4.55) and (4.56).

4.9 Summary

In this chapter, we generalized McCain's embedding of definite causal theories into logic programming. We expect that this work will provide a theoretical basis for extending the system COALA to more expressive action languages, including the modular action language MAD [70]. It is essential, from this perspective, that our translation is applicable to synonymy rules, because such rules are closely related to the main new feature of MAD, its **import** construct.

Our translation is not applicable to causal rules with quantifiers in the head. It may be possible to extend it to positive occurrences of existential quantifiers, since an existentially quantified formula can be thought of as an infinite disjunction. But the translation would be a formula with positive occurrences of existential quantifiers as well, and it is not clear how to turn such a formula into executable code.

In Chapter 5, we will to extend the translation described above to causal theories with explainable function symbols, which correspond to non-Boolean fluents in action languages. Since the definition of a stable model does not allow function symbols to be intensional, such a generalization will involve extending the language by auxiliary predicate symbols.

Chapter 5

Eliminating Functions from a Causal Theory

In Chapter 4 we generalized McCain's translation to a more general class of causal theories. The generalization allows us to reason about the domains where all fluents are Boolean-valued, which in causal theory is represented as predicate symbols, by translating the causal theory into logic program and calling ASP solvers to compute its models. However, the translation is not directly applicable to fluents with non-Boolean values, represented by function symbols, such as the location of an object in Example 2.3.6. There is a good reason for this limitation: describing non-Boolean fluents by logic programs involves an additional difficulty in view of the fact that rules in a logic program characterize predicates, not functions.[1]

Our approach is to show how a function symbol in a causal theory can be eliminated in favor of a new predicate symbol. In classical logic, this process is well understood. For instance, addition in first-order arithmetic can be described using a ternary predicate symbol, instead of a binary function symbol: we can write $sum(x, y, z)$ instead of $x + y = z$. (This alternative

[1]The language of [55] is not an exception: it does not permit formulas like $loc(x, t{+}1) = y$ in heads of rules. Explainable functions are allowed in more recent generalizations of the stable model semantics [5, 45].

leads to the use of cumbersome formulas, of course, when complex algebraic expressions are involved.) Extending this process to nonmonotonic causal logic is not straightforward, especially if we want to arrive eventually at an executable ASP program.

In this chapter, we will describe two procedures for eliminating function constants from a causal theory in favor of predicate constants, "general" and "definite." Then we will show how definite elimination can help us turn a causal theory into executable ASP code, and how it can be extended to rules that express the synonymy of function symbols.

5.1 Plain Causal Theories

Let f be a function constant. An atomic formula is *f-plain* if

- it does not contain f, or
- has the form $f(\mathbf{t}) = u$, where $\mathbf{t}$ is a tuple of terms not containing f, and u is a term not containing f.

A first-order formula, a causal rule, or a causal theory is *f-plain* if all its atomic subformulas are f-plain. For instance, the causal theory T_3 from Example 2.3.4 is c-plain, and the causal theory T_4 from Example 2.3.6 is loc-plain.

It is easy to transform any first-order formula into an equivalent f-plain formula. For instance, $p(f(f(x))$ is equivalent to the f-plain formula

$$\exists yz(f(x) = y \wedge f(y) = z \wedge p(z)).$$

Any causal theory can be transformed into an equivalent f-plain causal theory by applying this transformation to the heads and bodies of all rules. In Sections 5.2–5.6 we will assume that the given causal theory is f-plain.

5.2 General Elimination

Let T be an f-plain causal theory, where f is an explainable function constant. The causal theory T' is obtained from T as follows:

(1) in the signature of T, replace f with a new explainable predicate constant p of arity $n+1$, where n is the arity of f;

(2) in the rules of T, replace each subformula $f(\mathbf{t}) = u$ with $p(\mathbf{t}, u)$;

(3) add the rules

$$(\exists y)p(\mathbf{x}, y) \Leftarrow \top \tag{5.1}$$

and

$$\neg p(\mathbf{x}, y) \vee \neg p(\mathbf{x}, z) \Leftarrow y \neq z, \tag{5.2}$$

where $\mathbf{x}$ is a tuple of variables, and the variables $\mathbf{x}$, y, z are pairwise distinct.

Rule (5.2) expresses, in the language of causal logic, the uniqueness of y such that $p(\mathbf{x}, y)$.

Theorem 5.2.1. *The sentence*

$$\forall \mathbf{x} y (p(\mathbf{x}, y) \leftrightarrow f(\mathbf{x}) = y) \tag{5.3}$$

entails $T \leftrightarrow T'$.

Example 5.2.1 (Example 2.3.4, continued)**.** The result T_3' of applying this transformation to T_3 and to the object constant c as f is the causal theory

$$\begin{aligned} \bot &\Leftarrow a = b, \\ p(a) &\Leftarrow p(a), \\ p(b) &\Leftarrow q, \\ (\exists y)p(y) &\Leftarrow \top, \\ \neg p(y) \vee \neg p(z) &\Leftarrow y \neq z \end{aligned}$$

with the explainable symbol p. According to Theorem 5.2.1, the equivalence between T_3 and T_3' is entailed by the sentence

$$\forall y(p(y) \leftrightarrow c = y). \tag{5.4}$$

Example 5.2.2 (Example 2.3.6, continued)**.** The result T_4' of applying this transformation to T_4 and to the function constant *loc* as f is the causal theory

$$\begin{aligned} \bot &\Leftarrow 0 = 1, \\ \bot &\Leftarrow 0 = none, \\ \bot &\Leftarrow 1 = none, \\ obj(x) \wedge place(y) &\Leftarrow move(x, y) \\ at(x, 0, y) &\Leftarrow at(x, 0, y) \wedge obj(x) \wedge place(y), \\ at(x, 1, y) &\Leftarrow move(x, y), \\ at(x, 1, y) &\Leftarrow at(x, 0, y) \wedge at(x, 1, y) \wedge obj(x) \wedge place(y), \\ at(x, t, none) &\Leftarrow \neg obj(x), \\ at(x, t, none) &\Leftarrow t \neq 0 \wedge t \neq 1, \\ (\exists y)at(x, t, y) &\Leftarrow \top, \\ \neg at(x, t, y) \vee \neg at(x, t, z) &\Leftarrow y \neq z \end{aligned}$$

with the explainable symbol *at*. According to Theorem 5.2.1, the equivalence between T_4 and T_4' is entailed by the sentence

$$\forall xty(at(x, t, y) \leftrightarrow loc(x, t) = y) \tag{5.5}$$

By repeated applications of this process we can eliminate all explainable function symbols, provided that T is f-plain for each explainable symbol f.

The following corollary shows that there is a simple 1–1 correspondence between models of T and models of T'. Recall that the signature of T' is obtained from the signature of T by replacing f with p. For any interpretation I of the signature of T, by I^f_p we denote the interpretation of the signature of T' obtained from I by replacing the function f^I with the set p^I that consists of the tuples

$$\langle \xi_1, \ldots, \xi_n, f^I(\xi_1, \ldots, \xi_n) \rangle$$

for all $\xi_1, \ldots, \xi_n$ from the universe of I.

Corollary 5.2.2. *(a) An interpretation I of the signature of T is a model of T iff I^f_p is a model of T'. (b) An interpretation J of the signature of T' is a model of T' iff $J = I^f_p$ for some model I of T.*

Part (a) follows from the fact that the “union” of I and I^f_p satisfies (5.3). To show that any model of T' can be represented in the form I^f_p for some interpretation I, note that T' contains rules (5.1), (5.2) and consequently entails

$$\forall \mathbf{x}(\exists! y)p(\mathbf{x}, y). \tag{5.6}$$

5.3 Proof of the Theorem 5.2.1

In the following, assume T is a f-plain theory with a set of rules $F \Leftarrow G$ where F and G are first-order formulas, an explainable function symbol f, and a set of explainable function symbols $\mathbf{c}$ other than f. $T^{\dagger}(\upsilon f, \upsilon\mathbf{c})$ stands for

$$\bigwedge_{F \Leftarrow G \in T} \widetilde{\forall}_{obj} \left(G \rightarrow F^{f,\mathbf{c}}_{\upsilon f,\upsilon\mathbf{c}}\right)$$

T is the shorthand for the second-order sentence

$$\forall \upsilon f\, \upsilon\mathbf{c}(T^{\dagger}(\upsilon f, \upsilon\mathbf{c}) \leftrightarrow (\upsilon f = f) \wedge (\mathbf{c} = \upsilon\mathbf{c})) \tag{5.7}$$

Assume T' is a causal theory obtained from T by applying general elimination to function symbol f. Then $(T')^{\dagger}(\upsilon p, \upsilon\mathbf{c})$ stands for

$$\bigwedge_{F \Leftarrow G \in T} \widetilde{\forall}_{obj} \left(G^f_p \rightarrow (F^f_p)^{p,\mathbf{c}}_{\upsilon p,\upsilon\mathbf{c}}\right) \wedge UE$$

where G^f_p is the formula obtained from G by replacing $f(\mathbf{t}) = u$ by $p(\mathbf{t}, u)$, F^f_p is the formula obtained from F by replacing $f(\mathbf{t}) = u$ by $p(\mathbf{t}, u)$, and UE expresses that the conjunction of

$$\widetilde{\forall}(y \neq z \rightarrow \neg\upsilon p(\mathbf{x}, y) \vee \neg\upsilon p(\mathbf{x}, z)),$$
$$\widetilde{\forall}(\top \rightarrow (\exists y)\upsilon p(\mathbf{x}, y)).$$

Clearly, UE is equivalent to

$$(\forall\mathbf{x}\exists! y)\upsilon p(\mathbf{x}, y). \tag{5.8}$$

T' is the shorthand for the second-order sentence

$$\forall \upsilon p\, \upsilon\mathbf{c}((T')^{\dagger}(\upsilon p, \upsilon\mathbf{c}) \leftrightarrow (\upsilon p = p) \wedge (\upsilon\mathbf{c} = \mathbf{c})) \tag{5.9}$$

Theorem 5.2.1 is proved as follows.

Proof. ($\Rightarrow$) Assume (5.3), (5.7), that is

$$\forall \upsilon f\, \upsilon \mathbf{c} \left(\left(\bigwedge_{F \Leftarrow G \in T} \widetilde{\forall}_{obj} \left(G \rightarrow F^{f,\mathbf{c}}_{\upsilon f,\upsilon \mathbf{c}} \right) \right) \leftrightarrow (\upsilon f = f) \wedge (\upsilon \mathbf{c} = \mathbf{c}) \right), \tag{5.10}$$

our goal is to prove for any υp,

$$\left(\bigwedge_{F \Leftarrow G \in T} \widetilde{\forall}_{obj} \left(G^f_p \rightarrow (F^f_p)^{p,\mathbf{c}}_{\upsilon p,\upsilon \mathbf{c}} \right) \wedge UE \right) \leftrightarrow (\upsilon p = p) \wedge (\upsilon \mathbf{c} = \mathbf{c}). \tag{5.11}$$

Take υp. Consider two cases.

Case 1: $\neg UE$. Then the left-hand side of (5.11) is $\bot$. In the presence of (5.3), the right-hand side of (5.11) entails

$$\forall \mathbf{x} y (\upsilon p(\mathbf{x}, y) \leftrightarrow f(\mathbf{x}) = y)$$

which violates $\neg UE$. So (5.11) holds trivially.

Case 2: UE. Then there exists υf such that

$$\forall \mathbf{x} y (\upsilon f(\mathbf{x}) = y \leftrightarrow \upsilon p(\mathbf{x}, y)). \tag{5.12}$$

Under this condition, $\upsilon p = p$ can be rewritten as

$$\forall \mathbf{x} y (\upsilon f(\mathbf{x}) = y \leftrightarrow p(\mathbf{x}, y)),$$

The formula $(F^f_p)^{p,\mathbf{c}}_{\upsilon p,\upsilon \mathbf{c}}$ can be obtained from $F^{f,\mathbf{c}}_{\upsilon f,\upsilon \mathbf{c}}$ by replacing each subformula of the form $\upsilon f(\mathbf{t}) = u$ by $\upsilon p(\mathbf{t}, u)$, that is $(F^{f,\mathbf{c}}_{\upsilon f,\upsilon \mathbf{c}})^{\upsilon f}_{\upsilon p}$. From (5.12), $(F^f_p)^{p,\mathbf{c}}_{\upsilon p,\upsilon \mathbf{c}} \leftrightarrow F^{f,\mathbf{c}}_{\upsilon f,\upsilon \mathbf{c}}$. This equivalence, along with UE and 5.12, allow us to rewrite (5.11) as

$$\left(\bigwedge_{F\Leftarrow G\in T}\widetilde{\forall}_{obj}\left(G^f_p \to F^{f,\mathbf{c}}_{\upsilon f,\upsilon\mathbf{c}}\right)\right) \leftrightarrow (\forall \mathbf{x}y(\upsilon f(\mathbf{x}) = y \leftrightarrow p(\mathbf{x},y)) \land (\upsilon\mathbf{c} = \mathbf{c})). \tag{5.13}$$

In the presence of (5.3), G^f_p can be rewritten as G, and $\forall \mathbf{x}y(\upsilon f(\mathbf{x}) = y \leftrightarrow p(\mathbf{x},y))$ can be rewritten as

$$\forall \mathbf{x}y(\upsilon f(\mathbf{x}) = f(\mathbf{x})),$$

so that (5.13) follows from (5.10).

($\Leftarrow$) Assume (5.3) and T', that is (5.9), our goal is to prove, for any υf and $\upsilon\mathbf{c}$,

$$\left(\bigwedge_{F\Leftarrow G\in T}\widetilde{\forall}_{obj}\left(G \to F^{f,\mathbf{c}}_{\upsilon f,\upsilon\mathbf{c}}\right)\right) \leftrightarrow (\upsilon f = f) \land (\upsilon\mathbf{c} = \mathbf{c}) \tag{5.14}$$

Take υp satisfying (5.12). By (5.9),

$$\left(\bigwedge_{F\Leftarrow G\in T}\widetilde{\forall}_{obj}\left(G^f_p \to (F^f_p)^{p,\mathbf{c}}_{\upsilon p,\upsilon\mathbf{c}}\right) \land UE\right) \leftrightarrow (\upsilon p = p) \land (\upsilon\mathbf{c} = \mathbf{c}), \tag{5.15}$$

From (5.12) we can derive (5.8) and consequently UE. As above, using UE and (5.3), we can rewrite (5.15) as (5.14).

5.4 Definite Elimination

Unfortunately, the elimination process described in the previous section does not help us turn a causal theory with explainable function symbols into a logic program. The problem is that the translation from causal logic into logic programming described in Chapter 4 does not apply to causal rules with an existential quantifier in the head, such as (5.1). We will now describe an alternative elimination process, which is limited to causal theories of a special form but does not add rules containing quantifiers.

Consider an f-plain causal theory T, where f is an explainable function constant f satisfying the following condition: the head of any rule of T either does not contain f or has the form $f(\mathbf{t}) = u$, where $\mathbf{t}$ is a tuple of terms not containing explainable symbols, and u is a term not containing explainable symbols. The causal theory T'' is obtained from T as follows:

(1) in the signature of T, replace f with a new explainable predicate constant p of arity $n+1$, where n is the arity of f;

(2) in the rules of T, replace each subformula $f(\mathbf{t}) = u$ with $p(\mathbf{t}, u)$;

(3′) add the rule

$$\neg p(\mathbf{x}, y) \Leftarrow \neg p(\mathbf{x}, y), \tag{5.16}$$

where $\mathbf{x}$ is a tuple of variables, and the variables $\mathbf{x}$, y are pairwise distinct.

Rule (5.16) expresses the "closed world assumption" for p: by default, $p(\mathbf{x}, y)$ is false.

We call this process "definite elimination" because all new causal rules that it introduces are definite (Section 2.3.2). Definite elimination is applicable to the causal theories from Examples 1 and 2, but it cannot be applied, for instance, to a theory containing a rule with the head of the form $f(x) = y \vee f(x) = z$ or $\neg(f(x) = y)$.

Theorem 5.4.1. *The sentences (5.3) and*

$$\exists xy(x \neq y) \tag{5.17}$$

entail $T \leftrightarrow T''$.

Formula (5.17) expresses that the universe contains at least two elements. Without this assumption, the statement of Theorem 5.4.1 would be incorrect. Indeed, consider the causal theory T_4 with an explainable function symbol f that consists of the single rule $\top \Leftarrow \top$. It is easy to check that T_4 is equivalent to $\forall xy(x = y)$. On the other hand, T_4'' consists of the rules

$$\begin{gathered} \top \Leftarrow \top, \\ \neg p(\mathbf{x}, y) \Leftarrow \neg p(\mathbf{x}, y), \end{gathered}$$

and is equivalent to $\forall \mathbf{x} y \neg p(\mathbf{x}, y)$. The interpretation with a singleton universe that makes p identically true satisfies (5.3) and is a model of T_4, but it is not a model of T_4''.

Corollary 5.4.2. *If T contains a constraint of the form $\bot \Leftarrow t_1 = t_2$, where t_1, t_2 don't contain explainable function symbols, then (5.3) entails $T \leftrightarrow T''$.*

Indeed, if T contains the constraint $\bot \Leftarrow a = b$ then T'' contains it also, so that (5.17) is entailed both by T and by T''.

Example 5.4.1 (Example 2.3.4, continued)**.** The result T_3'' of applying definite elimination to T_3 and to the object symbol c is the theory

$$\begin{aligned} \bot &\Leftarrow a = b, \\ p(a) &\Leftarrow p(a), \\ p(b) &\Leftarrow q, \\ \neg p(y) &\Leftarrow \neg p(y) \end{aligned} \tag{5.18}$$

with the explainable symbol p. By Corollary 5.4.2, the equivalence between T_3 and T_3'' is entailed by sentence (5.4). Using the completion theorem (Fact 2.3.1), it is easy to check that causal theory (5.18) is equivalent to the first-order sentence

$$a \neq b \wedge (q \leftrightarrow p(b)) \wedge \forall y (p(y) \leftrightarrow (y = a \vee y = b)).$$

Under assumption (5.6), which can be written in this case as

$$(\exists! y) p(y), \tag{5.19}$$

this sentence can be equivalently transformed into a formula conveying the same information as (2.12):

$$a \neq b \wedge (q \rightarrow p(b)) \wedge (\neg q \rightarrow p(a)). \tag{5.20}$$

Example 5.4.2 (Example 2.3.6, continued)**.** The result T_4'' of applying definite elimination to theory T_4 and to function symbol *loc* is the theory

$$\begin{aligned}\bot &\Leftarrow 0 = 1,\\ \bot &\Leftarrow 0 = none,\\ \bot &\Leftarrow 1 = none,\\ obj(x) \wedge place(y) &\Leftarrow move(x, y)\\ at(x, 0, y) &\Leftarrow at(x, 0, y) \wedge obj(x) \wedge place(y),\\ at(x, 1, y) &\Leftarrow move(x, y)\\ at(x, 1, y) &\Leftarrow at(x, 0, y) \wedge at(x, 1, y) \wedge obj(x) \wedge place(y),\\ at(x, t, none) &\Leftarrow \neg obj(x),\\ at(x, t, none) &\Leftarrow t \neq 0 \wedge t \neq 1,\\ \neg at(x, t, y) &\Leftarrow \neg at(x, t, y)\end{aligned} \tag{5.21}$$

with the explainable symbol *at*. By Corollary 5.4.2, the equivalence between T_4 and T_4'' is entailed by sentence (5.5). Using the completion theorem from [44] we can show that under assumption (5.6), which can be written in this case as

$$\forall xt(\exists! y)\, at(x, t, y), \tag{5.22}$$

theory (5.21) can be equivalently transformed into the conjunction of the universal closures of the formulas

$$\begin{aligned}&\neg(0 = 1),\ \neg(0 = none),\ \neg(1 = none),\\ &\forall xy(at(x, 0, y) \wedge \neg obj(x) \to y = none),\\ &\forall xy(at(x, 0, y) \wedge obj(x) \to place(y)),\\ &\forall xy(at(x, 1, y) \leftrightarrow ((move(x, y) \wedge obj(x) \wedge place(y))\\ &\qquad\qquad \vee (at(x, 0, y) \wedge obj(x) \wedge place(y) \wedge \neg\exists w(move(x, w) \wedge place(w)))\\ &\qquad\qquad \vee (y = none \wedge \neg obj(x)))),\\ &\forall xyt((t \neq 0 \wedge t \neq 1) \to (at(x, t, y) \leftrightarrow y = none)).\end{aligned}$$

The equivalence with the left-hand side $at(x, 1, y)$ is similar to successor state axioms in the sense of [69].

By repeated applications of this process we can eliminate all explainable function symbols provided that

- T is f-plain for each explainable function symbol f, and
- the head of each rule of T containing an explainable function symbol f has the form $f(\mathbf{t}) = u$, where $\mathbf{t}$ is a tuple of terms not containing explainable symbols, and u is a term not containing explainable symbols.

We saw in Section 5.2 that the mapping $I \mapsto I^f_p$ is a 1–1 correspondence between the class of models of T and the class of models of T'. For definite elimination, this mapping establishes a 1–1 correspondence between the models of T with the universe of cardinality > 1 and the models of T'' with the universe of cardinality > 1 that satisfy additional condition (5.6); that condition, generally, is not entailed by T''. By $T^{\exists!}$ we denote the causal theory obtained from T'' by adding the constraint

$$\bot \Leftarrow (\exists! y) p(\mathbf{x}, y).$$

It is clear that $T^{\exists!}$ is equivalent to the conjunction of T'' with (5.6).

Corollary 5.4.3. *(a) An interpretation I of the signature of T with the universe of cardinality > 1 is a model of T iff I^f_p is a model of $T^{\exists!}$. (b) An interpretation J of the signature of $T^{\exists!}$ with the universe of cardinality > 1 is a model of $T^{\exists!}$ iff $J = I^f_p$ for some model I of T.*

For instance, the mapping $I \mapsto I^c_p$ establishes a 1–1 correspondence between the models of T_1 and the models of $T_1^{\exists!}$. Similarly, the mapping $I \mapsto I^{loc}_{at}$ establishes a 1–1 correspondence between the models of T_2 and the models of $T_2^{\exists!}$.

As discussed at the beginning of this section, the definite elimination process is limited to causal rules satisfying an additional syntactic restriction: if the head of a rule of T contains f then it should be an atomic formula. Without this restriction, the assertion of Theorem 5.4.1 would be incorrect. Consider, for instance, the causal theory T_5 with the rules

$$\begin{aligned} \bot &\Leftarrow a = b, \\ c = a \lor c = b &\Leftarrow \top \end{aligned}$$

and explainable c. It is easy to check that T_5 is inconsistent. On the other hand, T_5'' is

$$\begin{aligned} \bot &\Leftarrow a = b, \\ p(a) \lor p(b) &\Leftarrow \top, \\ \neg p(x) &\Leftarrow \neg p(x). \end{aligned}$$

This causal theory is equivalent to

$$a \neq b \land (\forall x(p(x) \leftrightarrow x = a) \lor \forall x(p(x) \leftrightarrow x = b)).$$

The interpretation with the universe $\{a, b\}$ that interprets c as a and p as $\{a\}$ satisfies (5.3), (5.17), and T_5'', but does not satisfy T_5.

Another restriction imposed at the beginning of this section is that in formulas $f(\mathbf{t}) = u$ in the heads of rules, $\mathbf{t}$ and u don't contain explainable symbols. Without this restriction, the assertion of Theorem 5.4.1 would be incorrect. Let T_6 be the causal theory obtained from T_5 by adding the rule

$$d = c \Leftarrow \top,$$

with both c and d explainable. It is easy to check that T_6 is inconsistent.

Consider the result T''_6 of applying definite elimination to T_6 and d:

$$\begin{aligned} \bot &\Leftarrow a = b, \\ c = a \lor c = b &\Leftarrow \top, \\ p(c) &\Leftarrow \top, \\ \neg p(x) &\Leftarrow \neg p(x), \end{aligned}$$

with p and c explainable. This causal theory is equivalent to

$$(a \neq b) \land (\forall x(p(x) \leftrightarrow x = a) \lor \forall x(p(x) \leftrightarrow x = b)) \land p(c).$$

The interpretation with the universe $\{a, b\}$ that interprets c as a, d as a, and p as $\{a\}$ satisfies (5.3), (5.17), and T''_6, but does not satisfy T_6.

5.5 Modified Definite Elimination

As discussed in Section 5.4, rule (5.16) expresses the closed world assumption for p. In "modified definite elimination," (5.16) is replaced by a definite counterpart of the uniqueness rule (5.2):

$$\neg p(\mathbf{x}, y) \Leftarrow p(\mathbf{x}, z) \land y \neq z$$

($\mathbf{x}$ is a tuple of variables, and the variables $\mathbf{x}$, y, z are pairwise distinct). We will denote the result of applying the modified definite elimination process to T by T'''. For instance, T'''_4 is

$$\begin{aligned} \bot &\Leftarrow a = b, \\ p(a) &\Leftarrow p(a), \\ p(b) &\Leftarrow q, \\ \neg p(y) &\Leftarrow p(z) \land y \neq z. \end{aligned}$$

Causal theories T'' and T''' are essentially equivalent to each other. To be precise, formula (5.3) entails $T'' \leftrightarrow T'''$. Indeed, $(T''')^\dagger$ can be obtained from $(T'')^\dagger$ by replacing

$$\forall \mathbf{x} y(\neg p(\mathbf{x}, y) \rightarrow \neg \upsilon p(\mathbf{x}, y)) \tag{5.23}$$

with

$$\forall \mathbf{x} y z(p(\mathbf{x}, z) \wedge y \neq z \rightarrow \neg \upsilon p(\mathbf{x}, y)). \tag{5.24}$$

Formula (5.24) can be rewritten as

$$\forall \mathbf{x} y(\exists z(p(\mathbf{x}, z) \wedge y \neq z) \rightarrow \neg \upsilon p(\mathbf{x}, y)).$$

In the presence of (5.3), the antecedent of this formula is equivalent to the antecedent of (5.23).

This fact implies that the assertions of Theorem 5.4.1 and Corollary 5.4.2 will remain valid if we replace T'' in their statements with T'''.

5.6 From Causal Logic to ASP

5.6.1 Turning Causal Theories into Logic Programs

The translation defined in Chapter 4 transforms a causal theory T satisfying some syntactic conditions into a first-order sentence F that has "the same meaning under the stable model semantics" as the theory T. (One of the conditions on T is that its explainable symbols are predicate constants, not function constants.) To be more precise, if the explainable symbols of T, along with the auxiliary predicate symbols introduced by the translation, are taken to be intensional then the stable models of F are identical to the models of T, provided that the interpretations of the auxiliary predicate symbols are "forgotten." In many cases, this translation can be applied to theories obtained by the definite elimination process described above.

Example 5.6.1 (Example 5.4.1, continued.)**.** Consider the causal theory $T_2^{\exists!}$, which, as we have seen, is "isomorphic" to T_2. It consists of the rules

$$\begin{aligned} \bot &\Leftarrow a = b, \\ p(a) &\Leftarrow p(a), \\ p(b) &\Leftarrow q, \\ \neg p(y) &\Leftarrow \neg p(y), \\ \bot &\Leftarrow \neg(\exists! y) p(y). \end{aligned}$$

The result of applying the translation from Chapter 4 to this theory is the conjunction of the universal closures of the formulas

$$\begin{aligned} \neg(a = b)&, \\ \neg\neg p(a) \to p(a)&, \\ \neg\neg q \to p(b)&, \\ \neg\neg\neg p(x) \to \widehat{p}(x)&, \\ \neg\neg\neg(\exists! y) p(y) \to \bot&, \\ \neg(p(x) \leftrightarrow \widehat{p}(x))&, \end{aligned} \tag{5.25}$$

where $\widehat{p}$ is an auxiliary predicate. Fact 2.4.3 shows that these formulas can be rewritten as

$$\begin{aligned} a \neq b&, \\ \neg\widehat{p}(a) \to p(a)&, \\ q \to p(b)&, \\ \neg p(x) \to \widehat{p}(x)&, \\ \neg\neg(\exists! y) p(y)&, \\ \neg(p(x) \leftrightarrow \widehat{p}(x))& \end{aligned} \tag{5.26}$$

without changing the class of stable models. Thus the stable models of (the conjunction of the universal closures of) formulas (5.26) turn into the models of $T_1^{\exists!}$ as soon as the interpretation of $\widehat{p}$ is dropped. It follows that the class of stable models of (5.26) is "isomorphic" to the class of models of T_1.

Example 5.6.2 (Example 5.4.2, continued)**.** The result of applying the translation from Chapter 4 to $T_2^{\exists!}$ becomes, after simplifications,

$$\begin{aligned}
0 \neq 1,\ 0 &\neq \mathit{none},\ 1 \neq \mathit{none},\\
\neg\widehat{at}(x,0,y) \wedge \mathit{obj}(x) \wedge \mathit{place}(y) &\rightarrow \mathit{at}(x,0,y),\\
\mathit{move}(x,y) &\rightarrow \mathit{place}(y)\\
\mathit{move}(x,y) &\rightarrow \mathit{obj}(x)\\
\mathit{move}(x,y) &\rightarrow \mathit{at}(x,1,y),\\
\neg\widehat{at}(x,0,y) \wedge \neg\widehat{at}(x,1,y) \wedge \mathit{obj}(x) \wedge \mathit{place}(y) &\rightarrow \mathit{at}(x,1,y),\\
\neg \mathit{at}(x,t,y) &\rightarrow \widehat{at}(x,t,y),\\
\neg \mathit{obj}(x) &\rightarrow \mathit{at}(x,t,\mathit{none}),\\
t \neq 0 \wedge t \neq 1 &\rightarrow \mathit{at}(x,t,\mathit{none}),\\
\neg\neg(\forall xt \exists! y)&\mathit{at}(x,t,y),\\
\neg(\widehat{at}(x,t,y) &\leftrightarrow \mathit{at}(x,t,y)).
\end{aligned} \tag{5.27}$$

The class of stable models of (5.27) is "isomorphic" to the class of models of T_2.

5.6.2 Turning Causal Theories into Executable Code

In many cases, answer set solvers such as CLINGO allow us to generate the Herbrand stable models of a given sentence. Consequently they can be sometimes used to generate models of causal theories.

Example 5.6.3 (Example 5.6.1, continued)**.** We would like to find all models of T_2 with the universe $\{a, b\}$ in which the constants a, b represent themselves. These models correspond to the Herbrand stable models of (5.26). We can find them by running CLINGO on the following input:

```
u(a;b).  #domain u(X).
{q}.
```

```
p(a) :- not -p(a).
p(b) :- q.
-p(X) :- not p(X).
:- not 1{p(Z):u(Z)}1.
:- not p(X), not -p(X).
```

The first line expresses that the universe `u` consists of `a` and `b`, and that `X` is a variable for arbitrary elements of `u`. The choice rule in the second line says that q can be assigned an arbitrary value. The other lines correspond to all formulas (5.26) except for $a \neq b$ (the unique name assumption is true in all Herbrand models and thus is taken by CLINGO for granted), with the classical negation `-p` representing $\widehat{p}$.[2]

Given this input, CLINGO generates two stable models: one containing `q` and `p(b)`, the other containing `p(a)`. Consequently T_1 has two models of the kind that we are interested in: in one of them q is true and the value of c is b; in the other, q is false and the value of c is a.

Example 5.6.4 (Example 5.6.2, continued.)**.** Consider the dynamic domain consisting of two objects o_1, o_2 that can be located in any of two places l_1, l_2. What are the possible locations of the objects after moving o_1 to l_2, for each

[2]The last line of (5.26) can be replaced by the pair of formulas

$$\neg(p(x) \wedge \widehat{p}(x)),\ \neg(\neg p(x) \wedge \neg\widehat{p}(x));$$

with $\widehat{p}$ represented by classical negation, the former is redundant.

possible initial state? To answer this question, we will find the models of T_2 with the universe

$$\{o_1, o_2, l_1, l_2, 0, 1, none\}$$

such that

- each of the constants 0, 1, *none* represents itself,
- the extent of *obj* is $\{o_1, o_2\}$,
- the extent of *place* is $\{l_1, l_2\}$, and
- the extent of *move* is $\{\langle o_1, l_2\rangle\}$.

To this end, we will find the stable models of (5.27) that satisfy all these conditions.

This computational task is equivalent to finding the Herbrand models of the sentence obtained by conjoining the universal closures of formulas (5.27) with the formulas

$$obj(o_1),\ obj(o_2),\ place(l_1),\ place(l_2),\ move(o_1, l_2)$$

(o_1, o_2, l_1, l_2 are new object constants), with *obj*, *place* and *move* included in the list of intensional predicates along with *at* and $\widehat{at}$.[3] To find these models, we run CLINGO on the following input:

[3]This claim can be justified using the splitting theorem (Fact 2.4.4).

```
u(o1;o2;l1;l2;0;1;none).
#domain u(X).  #domain u(T).  #domain u(Y).
obj(X) :- move(X,Y).
place(Y) :- move(X,Y).
at(X,0,Y) :- not -at(X,0,Y), obj(X), place(Y).
at(X,1,Y) :- move(X,Y).
at(X,1,Y) :- not -at(X,0,Y), not -at(X,1,Y), obj(X), place(Y).
-at(X,T,Y) :- not at(X,T,Y).
at(X,T,none) :- not obj(X).
at(X,T,none) :- T!=0, T!=1.
:- not 1{at(X,T,Z):u(Z)}1.
:- not at(X,T,Y), not -at(X,T,Y).
obj(o1;o2).  place(l1;l2).  move(o1,l2).
```

CLINGO generates 4 stable models, one for each possible combination of the locations of `o1` and `o2` at time 0. In every stable model, at time 1 object `o1` is at `l2`, and object `o2` is at the same place where it was at time 0.

5.7 Synonymy Rules

In this section we extend the definite elimination process to the case when several explainable function constants are eliminated in favor of predicate constants simultaneously, and the causal theory may contain rules of the form

$$f_1(\mathbf{t}^1) = f_2(\mathbf{t}^2) \Leftarrow G,$$

where f_1 and f_2 are two of the symbols that are eliminated. This rule expresses that there is a cause for $f_1(\mathbf{t}^1)$ to be "synonymous" to $f_2(\mathbf{t}^2)$ under condition G.

Consider a causal theory T and a tuple $\mathbf{f}$ of explainable function constants such that the bodies of the rules of T are f-plain for all members f of $\mathbf{f}$, and the head of any rule of T

- does not contain members of $\mathbf{f}$, or
- has the form $f(\mathbf{t}) = u$, where f is a member of $\mathbf{f}$, $\mathbf{t}$ is a tuple of terms not containing explainable symbols, and u is a term not containing explainable symbols, or
- has the form $f_1(\mathbf{t}^1) = f_2(\mathbf{t}^2)$, where f_1, f_2 are members of $\mathbf{f}$, and $\mathbf{t}^1$, $\mathbf{t}^2$ are tuples of terms not containing explainable symbols.

The causal theory T'' is obtained from T as follows:

(1) in the signature of T, replace each member f of $\mathbf{f}$ with a new explainable predicate constant p of arity $n+1$, where n is the arity of f;

(2a) in the rules of T, replace each subformula $f(\mathbf{t}) = u$ such that f is a member of $\mathbf{f}$ and u doesn't contain members of $\mathbf{f}$, with $p(\mathbf{t}, u)$;

(2b) in the heads of rules of T, replace each formula $f_1(\mathbf{t}^1) = f_2(\mathbf{t}^2)$ such that f_1, f_2 are members of $\mathbf{f}$, with $p_1(\mathbf{t}^1, y) \leftrightarrow p_2(\mathbf{t}^2, y)$, where y is a new variable;

(3′) add, for every new predicate p, the rule (5.16).

Theorem 5.7.1. *Sentences (5.3) for all f from* $\mathbf{f}$ *and sentence (5.17) entail* $T \leftrightarrow T''$.

Example 5.7.1. Consider the causal theory

$$\begin{aligned} f(x) = y &\Leftarrow a(x,y),\\ g(x) = y &\Leftarrow b(x,y),\\ f(x) = g(x) &\Leftarrow c(x) \end{aligned}$$

with the explainable symbols f, g. Its translation is

$$\begin{aligned} p(x,y) &\Leftarrow a(x,y),\\ q(x,y) &\Leftarrow b(x,y),\\ p(x,y) \leftrightarrow q(x,y) &\Leftarrow c(x),\\ \neg p(x,y) &\Leftarrow \neg p(x,y),\\ \neg q(x,y) &\Leftarrow \neg q(x,y) \end{aligned} \tag{5.28}$$

with the explainable symbols p, q.

Theorem 5.7.1 turns into Theorem 5.4.1 in the special case when $\mathbf{f}$ is a single function symbol and T does not contain synonymy rules.

By applying the transformation described in Section 4.4, the causal theory above can be translated into the conjunction of the universal closures of the following formulas:

$$\begin{aligned} \neg\neg a(x,y) &\rightarrow p(x,y),\\ \neg\neg b(x,y) &\rightarrow q(x,y),\\ \neg\neg c(x) \wedge p(x,y) &\rightarrow q(x,y),\\ \neg\neg c(x) \wedge q(x,y) &\rightarrow p(x,y),\\ \neg\neg c(x) \wedge \widehat{p}(x,y) &\rightarrow \widehat{q}(x,y),\\ \neg\neg c(x) \wedge \widehat{q}(x,y) &\rightarrow \widehat{p}(x,y),\\ \neg\neg\neg p(x,y) &\rightarrow \widehat{p}(x,y),\\ \neg\neg\neg q(x,y) &\rightarrow \widehat{q}(x,y),\\ \neg(p(x) &\leftrightarrow \widehat{p}(x)),\\ \neg(q(x) &\leftrightarrow \widehat{q}(x)). \end{aligned}$$

where p and q are intensional predicates.

5.8 Proof of Theorem 5.7.1

5.8.1 Leading Special Case

We begin with a special case that will be extended to the general proof of Theorem 5.7.1 in the next section. In this special case, we assume, first, that $\mathbf{f}$ is the list of *all* explainable symbols of T; second, that the head of every rule of T includes a member of $\mathbf{f}$. It follows that T consists of rules of the forms

$$f(\mathbf{t}) = u \Leftarrow G, \tag{5.29}$$

and

$$f(\mathbf{t}^1) = g(\mathbf{t}^2) \Leftarrow G, \tag{5.30}$$

where no explainable symbols are contained in $\mathbf{t}$, $\mathbf{t}^1$ or $\mathbf{t}^2$.

In the leading special case, T is shorthand for the formula

$$\forall \upsilon\mathbf{f}(T^\dagger(\upsilon\mathbf{f}) \leftrightarrow \upsilon\mathbf{f} = \mathbf{f})$$

where $\upsilon\mathbf{f}$ is the list of new function variables υf similar to the members f of $\mathbf{f}$, and $T^\dagger(\upsilon\mathbf{f})$ is

$$\bigwedge_{f(\mathbf{t})=u\Leftarrow G\,\in T} \widetilde{\forall}_{obj}(G \rightarrow \upsilon f(\mathbf{t}) = u) \wedge \bigwedge_{f(\mathbf{t}^1)=g(\mathbf{t}^2)\Leftarrow G\,\in T} \widetilde{\forall}_{obj}(G \rightarrow \upsilon f(\mathbf{t}^1) = \upsilon g(\mathbf{t}^2)). \tag{5.31}$$

Similarly, T'' is shorthand for

$$\forall \upsilon\mathbf{p}((T'')^\dagger(\upsilon\mathbf{p}) \leftrightarrow \upsilon\mathbf{p} = \mathbf{p})$$

where $\upsilon\mathbf{p}$ is the list of new predicate variables υp similar to the members p of $\mathbf{p}$, and $(T'')^{\dagger}(\upsilon\mathbf{p})$ is

$$\bigwedge_{f(\mathbf{t})=u\Leftarrow G\in T} \widetilde{\forall}_{obj}(G^{\mathbf{f}}_{\mathbf{p}} \rightarrow \upsilon p(\mathbf{t},u)) \\ \wedge \bigwedge_{f(\mathbf{t}^1)=g(\mathbf{t}^2)\Leftarrow G\in T} \widetilde{\forall}_{obj}(G^{\mathbf{f}}_{\mathbf{p}} \rightarrow (\upsilon p(\mathbf{t}^1,y) \leftrightarrow \upsilon q(\mathbf{t}^2,y))) \wedge (\upsilon\mathbf{p} \leq \mathbf{p}) \tag{5.32}$$

where $G^{\mathbf{f}}_{\mathbf{p}}$ is the formula obtained from G by replacing $f(\mathbf{t}) = u$ by $p(\mathbf{t},u)$, for each member f of $\mathbf{f}$.

In the rest of the proof, we will drop the expressions $f(\mathbf{t}) = u \Leftarrow G \in T$ and $f(\mathbf{t}^1) = g(\mathbf{t}^2) \Leftarrow G \in T$ in conjunctions over rules of T.

Define $\Lambda(\upsilon\mathbf{f})$ as the list of lambda expressions

$$\lambda\mathbf{x}y(\upsilon f(\mathbf{x}) = y)$$

for all members υf of $\upsilon\mathbf{f}$. Using this notation, we can rewrite the conjunction of formulas (5.3) for all f from $\mathbf{f}$ and the corresponding members p of $\mathbf{p}$ as

$$\Lambda(\mathbf{f}) = \mathbf{p}.$$

Lemma 5.8.1. $\Lambda(\mathbf{f}) = \mathbf{p} \models T^{\dagger}(\mathbf{f}) \leftrightarrow (T'')^{\dagger}(\mathbf{p})$.

Proof. $T^{\dagger}(\mathbf{f})$ is

$$\bigwedge \widetilde{\forall}(G \rightarrow f(\mathbf{t}) = u) \wedge \bigwedge \widetilde{\forall}(G \rightarrow f(\mathbf{t}^1) = g(\mathbf{t}^2)).$$

Since $\Lambda(\mathbf{f}) = \mathbf{p}$, this formula can be rewritten as

$$\bigwedge \widetilde{\forall}(G^{\mathbf{f}}_{\mathbf{p}} \rightarrow p(\mathbf{t},u)) \wedge \bigwedge \widetilde{\forall}(G^{\mathbf{f}}_{\mathbf{p}} \rightarrow (p(\mathbf{t}^1,y) \leftrightarrow q(\mathbf{t}^2,y))).$$

This formula is identical to $(T'')^{\dagger}(\mathbf{p})$.

If predicate symbols p and q have the same arity, we'll write $p \wedge q$ for

$$\lambda\mathbf{x}(p(\mathbf{x}) \wedge q(\mathbf{x}))$$

where $\mathbf{x}$ is a tuple of distinct variables; $(p_1, \dots, p_n) \wedge (q_1, \dots, q_n)$ will stand for

$$(p_1 \wedge q_1, \dots, p_n \wedge q_n),$$

and similarly for predicate expressions in place of p, q.

Lemma 5.8.2. $\Lambda(\mathbf{f}) = \mathbf{p} \wedge T'' \models \forall\mathbf{f}(T^{\dagger}(\upsilon\mathbf{f}) \rightarrow \upsilon\mathbf{f} = \mathbf{f})$

Proof. From T'' we derive

$$\bigwedge \widetilde{\forall}(G^{\mathbf{f}}_{\mathbf{p}} \rightarrow p(\mathbf{t}, u)) \wedge \bigwedge \widetilde{\forall}(G^{\mathbf{f}}_{\mathbf{p}} \rightarrow (p(\mathbf{t}^1, y) \leftrightarrow q(\mathbf{t}^2, y))) \tag{5.33}$$

and

$$\begin{aligned}\forall\upsilon\mathbf{p}\Big(\Big(&\bigwedge \widetilde{\forall}_{obj}(G^{\mathbf{f}}_{\mathbf{p}} \rightarrow \upsilon p(\mathbf{t}, u)) \\ &\wedge \bigwedge \widetilde{\forall}_{obj}(G^{\mathbf{f}}_{\mathbf{p}} \rightarrow (\upsilon p(\mathbf{t}^1, y) \leftrightarrow \upsilon q(\mathbf{t}^2, y))) \wedge (\upsilon\mathbf{p} \le \mathbf{p})\Big) \rightarrow \upsilon\mathbf{p} = \mathbf{p}\Big).\end{aligned} \tag{5.34}$$

Assume $T^{\dagger}(\upsilon\mathbf{f})$, that is (5.31), The goal is to prove $\upsilon\mathbf{f} = \mathbf{f}$. Since $\Lambda(\mathbf{f}) = \mathbf{p}$, (5.33) and (5.34) can be rewritten as

$$\bigwedge \widetilde{\forall}(G \rightarrow f(\mathbf{t}) = u) \wedge \bigwedge \widetilde{\forall}(G \rightarrow (f(\mathbf{t}^1) = g(\mathbf{t}^2))) \tag{5.35}$$

and

$$\begin{aligned}\forall\upsilon\mathbf{p}\Big(\Big(&\bigwedge \widetilde{\forall}_{obj}(G \rightarrow \upsilon p(\mathbf{t}, u)) \\ &\wedge \bigwedge \widetilde{\forall}_{obj}(G \rightarrow (\upsilon p(\mathbf{t}^1, y) \leftrightarrow \upsilon q(\mathbf{t}^2, y))) \wedge (\upsilon\mathbf{p} \le \mathbf{p})\Big) \rightarrow \upsilon\mathbf{p} = \mathbf{p}\Big).\end{aligned} \tag{5.36}$$

Apply (5.36) to the predicates $\upsilon\mathbf{p}$ defined by the condition

$$\forall \mathbf{x} y(\upsilon p(\mathbf{x}, y) \leftrightarrow (f(\mathbf{x}) = y \wedge \upsilon f(\mathbf{x}) = y)). \tag{5.37}$$

The antecedent of (5.36) becomes the conjunction of

$$\bigwedge \widetilde{\forall}_{obj}(G \to f(\mathbf{t}) = u \wedge \upsilon f(\mathbf{t}) = u), \tag{5.38}$$

$$\bigwedge \widetilde{\forall}_{obj}(G \to ((\upsilon f(\mathbf{t}^1) = y \wedge f(\mathbf{t}^1) = y) \leftrightarrow (\upsilon g(\mathbf{t}^2) = y \wedge g(\mathbf{t}^2) = y))), \tag{5.39}$$

and

$$(\Lambda(\mathbf{f}) \wedge \Lambda(\upsilon\mathbf{f})) \leq \mathbf{p}. \tag{5.40}$$

We will show that (5.38), (5.39), (5.40) can be derived from (5.31) (5.35) and (5.37). Formula (5.38) follows from the first conjunctive term of (5.35) and the first conjunctive term of (5.31). Formula (5.39) follows from the second conjunctive term of (5.35) and the second conjunctive term of (5.31). Formula (5.40) follows from $\Lambda(\mathbf{f}) = \mathbf{p}$.

We have proved the antecedent of (5.36), so we can derive the consequent of (5.36). Since $\Lambda(\mathbf{f}) = \mathbf{p}$, the consequent can be rewritten as

$$(\Lambda(\mathbf{f}) \wedge \Lambda(\upsilon\mathbf{f})) = \Lambda(\mathbf{f}),$$

that is,

$$\bigwedge_{f \in \mathbf{f}} \forall \mathbf{x} y((f(\mathbf{x}) = y \wedge \upsilon f(\mathbf{x}) = y) \leftrightarrow f(\mathbf{x}) = y).$$

This formula is equivalent to

$$\bigwedge_{f \in \mathbf{f}} \forall \mathbf{x} y(f(\mathbf{x}) = y \to \upsilon f(\mathbf{x}) = y)$$

and consequently to $\mathbf{f} = \upsilon\mathbf{f}$.

Lemma 5.8.3. $\Lambda(\mathbf{f}) = \mathbf{p} \wedge \exists x_1 x_2 (x_1 \neq x_2) \wedge T \models (\forall \upsilon\mathbf{p})((T'')^\dagger(\upsilon\mathbf{p}) \rightarrow \upsilon\mathbf{p} = \mathbf{p})$.

Proof. From T,

$$\bigwedge \widetilde{\forall}(G \rightarrow f(\mathbf{t}) = u) \wedge \bigwedge \widetilde{\forall}(G \rightarrow f(\mathbf{t}^1) = g(\mathbf{t}^2)) \tag{5.41}$$

and

$$\forall \upsilon\mathbf{f} \left(\bigwedge \widetilde{\forall}_{obj}(G \rightarrow \upsilon f(\mathbf{t}) = u) \wedge \bigwedge \widetilde{\forall}_{obj}(G \rightarrow \upsilon f(\mathbf{t}^1) = \upsilon g(\mathbf{t}^2)) \rightarrow \mathbf{f} = \upsilon\mathbf{f} \right). \tag{5.42}$$

Take any $\upsilon\mathbf{p}$ satisfying $(T'')^\dagger(\upsilon\mathbf{p})$, that is, (5.32). Our goal is to show $\upsilon\mathbf{p} = \mathbf{p}$.

Under the condition $\Lambda(\mathbf{f}) = \mathbf{p}$, (5.32) can be rewritten as

$$\bigwedge \widetilde{\forall}_{obj}(G \rightarrow \upsilon p(\mathbf{t}, u)) \wedge \bigwedge \widetilde{\forall}_{obj}(G \rightarrow (\upsilon p(\mathbf{t}^1, y) \leftrightarrow \upsilon q(\mathbf{t}^2, y))) \wedge (\upsilon\mathbf{p} \leq \mathbf{p}). \tag{5.43}$$

The third conjunctive term shows that we only need to prove $\mathbf{p} \leq \upsilon\mathbf{p}$, or, equivalently, $\Lambda(\mathbf{f}) \leq \upsilon\mathbf{p}$. This is a conjunction of several formulas, one for each member of the tuple $\mathbf{f}$, and, to simplify notation, we will show how to prove the first conjunctive term

$$(\forall \mathbf{x}) \upsilon p_1(\mathbf{x}, f_1(\mathbf{x})).$$

Assume that for some $\mathbf{x}^*$, $\neg \upsilon p_1(\mathbf{x}^*, f_1^*(\mathbf{x}^*))$. Then we are going to define $\upsilon\mathbf{f}$ such that the antecedent of (5.42) is satisfied, while the consequent is violated; that will complete the proof of the lemma.

For each member f of $\mathbf{f}$, choose a new predicate variable δp of the same arity as f (so that the arity of δp is less by 1 than the arity of p.) By $\delta\mathbf{p}$

we denote the tuple consisting of all these variables. By $H_f(\mathbf{x})$ we denote the formula

$$\forall \delta \mathbf{p} \left(\delta p_1(\mathbf{x}^*) \wedge \left(\bigwedge \widetilde{\forall}_{obj}(G \to (\delta p(\mathbf{t}^1) \leftrightarrow \delta q(\mathbf{t}^2))) \right) \to \delta p(\mathbf{x}) \right).$$

In view of the assumption $\exists x_1 x_2 (x_1 \neq x_2)$, there exists a y^* such that

$$y^* \neq f_1(x^*).$$

Define each $\upsilon f \in \upsilon \mathbf{f}$ by the condition:

$$\forall \mathbf{x}((H_f(\mathbf{x}) \to \upsilon f(\mathbf{x}) = y^*) \wedge (\neg H_f(\mathbf{x}) \to \upsilon f(\mathbf{x}) = f(\mathbf{x}))). \tag{5.44}$$

The proof of the fact that $\upsilon \mathbf{f}$ is indeed a counterexample to (5.42) is based on two claims.

Claim 1: $\upsilon \mathbf{p} \leq \Lambda(\upsilon \mathbf{f})$.

To prove Claim 1, take any member υp^* of $\upsilon \mathbf{p}$ and consider any $\mathbf{x}^{**}$ and y^{**} such that

$$\upsilon p^*(\mathbf{x}^{**}, y^{**}). \tag{5.45}$$

We need to check that $\upsilon f^*(x^{**}) = y^{**}$. Since $\upsilon \mathbf{p} \leq \mathbf{p} = \Lambda(\mathbf{f})$,

$$f^*(x^{**}) = y^{**} \tag{5.46}$$

It remains to prove that $f^*(\mathbf{x}^{**}) = \upsilon f^*(\mathbf{x}^{**})$. To this end, we will show that $\neg H_{f^*}(\mathbf{x}^{**})$. We will define a set of predicates $\delta \mathbf{p}$ such that

$$\delta p_1(\mathbf{x}^*) \wedge \left(\bigwedge \widetilde{\forall}_{obj}(G \to (\delta p(\mathbf{t}^1) \leftrightarrow \delta q(\mathbf{t}^2))) \right), \tag{5.47}$$

but

$$\neg \delta p^*(\mathbf{x}^{**}). \tag{5.48}$$

Define each δp by the condition

$$\forall \mathbf{x}(\delta p(\mathbf{x}) \leftrightarrow \neg \upsilon p(\mathbf{x}, f(\mathbf{x}))). \tag{5.49}$$

By (5.45) and (5.46), $\upsilon p^*(\mathbf{x}^{**}, f(\mathbf{x}^{**}))$, so that (5.48) follows.

To prove (5.47), note first that by the choice of x^*, $\neg \upsilon p_1(\mathbf{x}^*, f_1(\mathbf{x}^*))$, so the first conjunctive term of (5.47) follows. By (5.43),

$$\widetilde{\forall}_{obj}(G \to (\upsilon p(\mathbf{t}^1, y) \leftrightarrow \upsilon q(\mathbf{t}^2, y))),$$

and consequently

$$\widetilde{\forall}_{obj}(G \to (\upsilon p(\mathbf{t}^1, f(\mathbf{t}^1)) \leftrightarrow \upsilon q(\mathbf{t}^2, f(\mathbf{t}^1)))).$$

By (5.41), it follows that

$$\bigwedge \widetilde{\forall}_{obj}(G \to (\upsilon p(\mathbf{t}^1, f(\mathbf{t}^1)) \leftrightarrow \upsilon q(\mathbf{t}^2, g(\mathbf{t}^2)))),$$

In combination with (5.49), this formula gives us the second conjunctive term of (5.47).

Claim 2: *For any conjunctive term*

$$\widetilde{\forall}(G^* \to f^*(\boldsymbol{t}^1) = g^*(\boldsymbol{t}^2))$$

of (5.41), sentence

$$\widetilde{\forall}(G^* \to (H_{f^*}(\boldsymbol{t}^1) \leftrightarrow H_{g^*}(\boldsymbol{t}^2)))$$

is logically valid.

$H_{f^*}(\mathbf{t}^1)$ is

$$\forall \delta\mathbf{p}\left(\delta p_1(\mathbf{x}^*) \wedge \bigwedge \widetilde{\forall}_{obj}(G \rightarrow (\delta p(\mathbf{t}^1) \leftrightarrow \delta q(\mathbf{t}^2))) \rightarrow \delta p^*(\mathbf{t}^1)\right). \qquad (5.50)$$

Under assumption G^*, from the antecedent of this formula we can derive

$$\delta p^*(\mathbf{t}^1) \leftrightarrow \delta q^*(\mathbf{t}^2)$$

so that (5.50) can be equivalently rewritten as

$$\forall \delta\mathbf{p}\left(\delta p_1^*(\mathbf{x}^*) \wedge \bigwedge \widetilde{\forall}_{obj}(G \rightarrow (\delta p(\mathbf{t}^1) \leftrightarrow \delta q(\mathbf{t}^2))) \rightarrow \delta q^*(\mathbf{t}^2)\right),$$

that is, $H_{g^*}(\mathbf{t}^2)$.

Now we return to the proof of Lemma 5.8.3. For $\upsilon\mathbf{f}$ defined by (5.44), we will prove that the antecedent of (5.42) is satisfied while the consequent not. From the definition of $H_f(\mathbf{x})$, it is clear that $H_{f_1}(\mathbf{x}^*)$. By the choice of υf, it follows that $\upsilon f_1(\mathbf{x}^*) = y^*$. By the choice of y^*, we conclude that $\upsilon f_1(\mathbf{x}^*) \neq f_1(\mathbf{x}^*)$, which contradicts the consequent of (5.42). So it remians to show that the antecedent of (5.42) is satisfied.

By Claim 1 and the first conjunctive term of (5.43), the first conjunctive term of (5.42) is satisfied. The remaining conjunctive terms have the form

$$\widetilde{\forall}_{obj}(G^* \rightarrow \upsilon f^*(\mathbf{t}^1) = \upsilon g^*(\mathbf{t}^2)).$$

Assume G^*. Case 1: $H_{f^*}(\mathbf{t}^1)$. Then by Claim 2, $H_{g^*}(\mathbf{t}^2)$. By the choice of υf, it follows that $\upsilon f^*(\mathbf{t}^1) = y^*$ and $\upsilon g^*(\mathbf{t}^2) = y^*$. Case 2: $\neg H_{f^*}(\mathbf{t}^1)$. Then by

Claim 2, $\neg H_{g^*}(\mathbf{t}^2)$. By the choice of υf, it follows that $\upsilon f^*(\mathbf{t}^1) = f^*(\mathbf{t}^1)$ and $\upsilon g^*(\mathbf{t}^2) = g^*(\mathbf{t}^2)$. It remains to notice that $f^*(\mathbf{t}^1) = g^*(\mathbf{t}^2)$ by (5.41).

The statement of Theorem 5.7.1 for the special case immediately follows from Lemmas 5.8.1–5.8.3.

5.8.2 General Case

To prove Theorem 5.7.1 in full generality, we divide a causal theory T into two parts

- theory T_1 containing a set of rules of form (5.29) and (5.30), with the explainable symbols of each rule in T_2 being those explainable in the same rule of T; and
- theory T_2 containing other rules, with explainable symbols of each rule in T_2 being those explainable in the same rule of T.

Clearly, T_1, T_2 are pairwise disjoint theories. Similarly, we divide T'' into two parts

- theory T_1'' containing a set of rules of corresponding to the rule (5.29) and (5.30) in T_1, and the set of explainable symbols in T_1'' corresponds to those in T_1.
- theory T_2'' containing other rules. The explainable symbols of each rule in T_2 are those explainable in the same rule in T''.

Clearly, T_1'', T_2'' are pairwise disjoint theories. By [44, Lemma 1],

$$T \leftrightarrow T_1 \wedge T_2$$

and

$$T'' \leftrightarrow T_1'' \wedge T_2''$$

By Theorem 5.4.1,

$$\mathbf{p} = \Lambda(\mathbf{f}) \wedge \exists x_1 x_2 (x_1 \neq x_2) \models T_2 \leftrightarrow T_2''$$

By Lemma 5.8.1,

$$\mathbf{p} = \Lambda(\mathbf{f}) \wedge \exists x_1 x_2 (x_1 \neq x_2) \models T_1 \leftrightarrow T_1''$$

Therefore, we obtain

$$\mathbf{p} = \Lambda(\mathbf{f}) \wedge \exists x_1 x_2 (x_1 \neq x_2) \models T_1 \wedge T_2 \leftrightarrow T_1'' \wedge T_2''$$

and finally

$$\mathbf{p} = \Lambda(\mathbf{f}) \wedge \exists x_1 x_2 (x_1 \neq x_2) \models T \leftrightarrow T''.$$

5.9 Related Work

The problem addressed in this chapter is similar to the problem of eliminating multi-valued propositional constants from a multi-valued causal theory [33]. In this sense, our general elimination and modified definite elimination are similar to the elimination methods proposed in [38, Section 6.4.2]. On the other hand, modified definite elimination does not introduce rules similar to constraint (6.26) from [38], and our proofs (not included in this note) are entirely different: the semantics of multi-valued propositional constants is based on a fixpoint construction and does not refer to syntactic transformations.

Eliminating function constants in the framework of a different nonmonotonic formalism—a version of the stable model semantics—is discussed in [55].

To sum up, in this chapter we investigated some of the cases when an explainable function symbol can be eliminated from a first-order causal theory in favor of a predicate symbol. This is a step towards the goal of creating a compiler from the modular action language MAD [47] into answer set programming. It will differ from the current version of COALA [24] in that it will be applicable to action descriptions that involve non-Boolean fluents and synonymy rules. Translating causal rules with equivalences in the head, such as the third rule of (5.28), into ASP is studied (for the propositional case) in [39].

Chapter 6

Action Language $\mathcal{BC}$

In this chapter, we design a new action language based on action languages $\mathcal{B}$ and $\mathcal{C}$. The languages $\mathcal{B}$ and $\mathcal{C}$ have significant common core [31]. Nevertheless, some expressive possibilities of $\mathcal{B}$ are difficult or impossible to simulate in $\mathcal{C}$, and the other way around. The main advantage of $\mathcal{B}$ is that it allows the user to give Prolog-style recursive definitions. Recursively defined concepts, such as the reachability of a node in a graph, play important role in applications of automated reasoning about actions, including the design of the decision support system for the space shuttle [65]. On the other hand, the language $\mathcal{B}$, like STRIPS and ADL, solves the frame problem by incorporating the commonsense law of inertia in its semantics, which makes it difficult to talk about fluents whose behavior is described by defaults other than inertia. The position of a moving pendulum, for instance, is a non-inertial fluent: it changes by itself, and an action is required to prevent the pendulum from moving. The amount of liquid in a leaking container (Example 2.2.4) changes by itself, and an action is required to prevent it from decreasing. A spring-loaded door closes by itself, and an action is required to keep it open. Work on the action language $\mathcal{C}$ and its extension $\mathcal{C}$+ was partly motivated by examples of this kind. In these languages, the inertia assumption is expressed by axioms that the user is free to include or not to include. Other default assumptions about the relationship between the values of a fluent at different time instants can be postulated as well. On the other hand, some recursive definitions cannot be easily expressed in $\mathcal{C}$ and $\mathcal{C}+$.

The new action description language we define in this chapter, called $\mathcal{BC}$, combines the attractive features of $\mathcal{B}$ and $\mathcal{C}+$. This language, like $\mathcal{B}$, can be implemented using computational methods of answer set programming.

The main difference between $\mathcal{B}$ and $\mathcal{BC}$ is similar to the difference between inference rules and default rules. Informally speaking, a default rule allows us to derive its conclusion from its premise if its justification can be consistently assumed; default logic [68] makes this idea precise. In the language $\mathcal{B}$, a static law has the form

$$<\text{conclusion}> \ \textbf{if} \ <\text{premise}> .$$

In $\mathcal{BC}$, a static law may include a justification:

$$<\text{conclusion}> \ \textbf{if} \ <\text{premise}> \ \textbf{ifcons} \ <\text{justification}>$$

(**ifcons** is an acronym for "if consistent"). Dynamic laws may include justifications also.

The semantics of $\mathcal{BC}$ is defined by transforming action descriptions into logic programs under the stable model semantics. When static and dynamic laws of the language $\mathcal{B}$ are translated into the language of logic programming, as in [4], the rules that we get do not contain negation as failure. Logic programs corresponding to $\mathcal{B}$-descriptions do contain negation as failure, but this is because inertia rules are automatically included in them. In the case of $\mathcal{BC}$, on the other hand, negation as failure is used for translating justifications in both static and dynamic laws.

We define here two translations from $\mathcal{BC}$ into logic programming. One of them uses strong negation, and the other doesn't, but we show that both translations give the same meaning to $\mathcal{BC}$-descriptions.

Examples of formalizing commonsense domains discussed in this paper illustrate the expressive capabilities of $\mathcal{BC}$ and the use of answer set solvers for the automation of reasoning about actions described in this language. We state also two theorems relating $\mathcal{BC}$ to $\mathcal{B}$ and to $\mathcal{C}+$.

6.1 Syntax

An action description in the language $\mathcal{BC}$ includes a finite set of symbols of two kinds, *fluent constants* and *action constants*. Fluent constants are further divided into *regular* and *statically determined*. A finite set of cardinality ≥ 2, called the *domain*, is assigned to every fluent constant.

An *atom* is an expression of the form $f = v$, where f is a fluent constant, and v is an element of its domain. If the domain of f is $\{\mathbf{f}, \mathbf{t}\}$ then we say that f is *Boolean*.

A *static law* is an expression of the form

$$A_0 \textbf{ if } A_1, \ldots, A_m \textbf{ ifcons } A_{m+1}, \ldots, A_n \tag{6.1}$$

$(n \geq m \geq 0)$, where each A_i is an atom. It expresses, informally speaking, that every state satisfies A_0 if it satisfies $A_1, \dots, A_m$, and $A_{m+1}, \dots, A_n$ can be consistently assumed. If $m = 0$ then we will drop **if**; if $m = n$ then we will drop **ifcons**.

A *dynamic law* is an expression of the form

$$A_0 \textbf{ after } A_1, \dots, A_m \textbf{ ifcons } A_{m+1}, \dots, A_n \tag{6.2}$$

$(n \geq m \geq 0)$, where

- A_0 is an atom containing a regular fluent constant,
- each of $A_1, \dots, A_m$ is an atom or an action constant, and
- $A_{m+1}, \dots, A_n$ are atoms.

It expresses, informally speaking, that the end state of any transition satisfies A_0 if its beginning state and its action satisfy $A_1, \dots, A_m$, and $A_{m+1}, \dots, A_n$ can be consistently assumed about the end state. If $m = n$ then we will drop **ifcons**.

For any action constant a and atom A,

$$a \textbf{ causes } A$$

stands for

$$A \textbf{ after } a.$$

For any action constant a and atoms $A_0, \dots, A_m$ $(m > 0)$,

$$a \textbf{ causes } A_0 \textbf{ if } A_1, \dots, A_m$$

stands for

$$A_0 \textbf{ after } a, A_1, \ldots, A_m.$$

An *action description* in the language $\mathcal{BC}$ is a finite set consisting of static and dynamic laws.

6.2 Defaults and Inertia

Static laws of the form

$$A_0 \textbf{ if } A_1, \ldots, A_m \textbf{ ifcons } A_0 \tag{6.3}$$

and dynamic laws of the form

$$A_0 \textbf{ after } A_1, \ldots, A_m \textbf{ ifcons } A_0 \tag{6.4}$$

will be particularly useful. They are similar to normal defaults in the sense of [68]. We will write (6.3) as

$$\textbf{default } A_0 \textbf{ if } A_1, \ldots, A_m,$$

and we will drop **if** when $m = 0$. We will write (6.4) as

$$\textbf{default } A_0 \textbf{ after } A_1, \ldots, A_m.$$

For any regular fluent constant f, the set of the dynamic laws

$$\textbf{default } f = v \textbf{ after } f = v$$

for all v in the domain of f expresses the commonsense law of inertia for f. We will denote this set by

$$\textbf{inertial } f. \tag{6.5}$$

6.3 Semantics

For every action description D, we will define a sequence of logic programs $PN_0(D), PN_1(D), \dots$ so that the stable models of $PN_l(D)$ represent paths of length l in the transition system corresponding to D. The signature $\sigma_{D,l}$ of $PN_l(D)$ consists of

- expressions $i\!:\!A$ for nonnegative integers $i \leq l$ and all atoms A, and
- expressions $i\!:\!a$ for nonnegative integers $i < l$ and all action constants a.

Thus every element of the signature $\sigma_{D,l}$ is a "time stamp" i followed by an atom in the sense of Section 6.1 or by an action constant. The program consists of the following rules:

- the translations

  $$i\!:\!A_0 \leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \wedge \neg\neg\; i\!:\!A_{m+1} \wedge \cdots \wedge \neg\neg\; i\!:\!A_n$$

  $(i \leq l)$ of all static laws (6.1) from D,

- the translations

  $$(i+1)\!:\!A_0 \leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \wedge \neg\neg\; (i+1)\!:\!A_{m+1} \wedge \cdots \wedge \neg\neg\; (i+1)\!:\!A_n$$

  $(i < l)$ of all dynamic laws (6.2) from D,

- the choice rule $\{0 : A\}$ for every atom A containing a regular fluent constant,

- the choice rule $\{i\!:\!a\}$ for every action constant a and every $i < l$,
- the existence of value constraint

  $$\leftarrow \neg\, i\!:\!(f\!=\!v_1) \wedge \cdots \wedge \neg\, i\!:\!(f\!=\!v_k)$$

  for every fluent constant f and every $i \leq l$, where $v_1, \ldots, v_k$ are all elements of the domain of f,
- the uniqueness of value constraint

  $$\leftarrow i\!:\!(f\!=\!v) \wedge i\!:\!(f\!=\!w)$$

  for every fluent constant f, every pair of distinct elements v, w of its domain, and every $i \leq l$.

The transition system $T(D)$ represented by an action description D is defined as follows. For every stable model X of $PN_0(D)$, the set of atoms A such that $0\!:\!A$ belongs to X is a state of $T(D)$. In view of the existence of value and uniqueness of value constraints, for every state s and every fluent constant f there exists exactly one v such that $f\!=\!v$ belongs to s; this v is considered the value of f in state s. For every stable model X of $PN_1(D)$, $T(D)$ includes the transition $\langle s_0, \alpha, s_1 \rangle$, where s_i $(i = 0, 1)$ is the set of atoms A such that $i\!:\!A$ belongs to X, and α is the set of action constants a such that $0\!:\!a$ belongs to X.

The soundness of this definition is guaranteed by the following fact:

Theorem 6.3.1. *For every transition $\langle s_0, \alpha, s_1 \rangle$, s_0 and s_1 are states.*

The proofs of this theorem and other theorems stated in this chapter are given in Chapter 7.

We promised that stable models of $PN_l(D)$ would represent paths of length l in the transition system corresponding to D. For $l = 0$ and $l = 1$, this is clear from the definition of $T(D)$; for $l > 1$ this needs to be verified. For every set X of elements of the signature $\sigma_{D,l}$, let X^i $(i < l)$ be the triple consisting of

- the set of atoms A such that $i\!:\!A$ belongs to X,
- the set of action constants a such that $i\!:\!a$ belongs to X, and
- the set of atoms A such that $(i+1)\!:\!A$ belongs to X.

Theorem 6.3.2. *For every $l \geq 1$, X is a stable model of $PN_l(D)$ iff $X^0, \ldots, X^{l-1}$ are transitions.*

The rules contributed to $PN_l(D)$ by static law (6.3) have the form

$$i\!:\!A_0 \leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \wedge \neg\neg\, i\!:\!A_0.$$

They can be equivalently rewritten as

$$\{i\!:\!A_0\} \leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m$$

(see [46]). Similarly, the rules contributed to $PN_l(D)$ by dynamic law (6.4) have the form

$$(i+1)\!:\!A_0 \leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \wedge \neg\neg\,(i+1)\!:\!A_0.$$

They can be equivalently rewritten as

$$\{(i+1):A_0\} \leftarrow i:A_1 \wedge \cdots \wedge i:A_m.$$

In particular, the rules contributed by the commonsense law of inertia (6.5) can be rewritten as

$$\{(i+1):f=v\} \leftarrow i:f=v.$$

6.4 Other Abbreviations

In $\mathcal{BC}$-descriptions that involve Boolean fluent constants we will use abbreviations similar to those established for multi-valued formulas in [33, Section 2.1]: if f is Boolean then we will write the atom $f=\mathbf{t}$ as f, and the atom $f=\mathbf{f}$ as $\sim f$.

A *static constraint* is a pair of static laws of the form

$$\begin{array}{l} f=v \textbf{ if } A_1,\dots,A_m \\ f=w \textbf{ if } A_1,\dots,A_m \end{array} \tag{6.6}$$

where $v \neq w$ and $m>0$. We will write (6.6) as

$$\textbf{impossible } A_1,\dots,A_m.$$

The use of this abbreviation depends on the fact that the choice of f, v, and w in (6.6) is inessential, in the sense of Theorem 6.4.1 below. About action descriptions D_1 and D_2 we say that they are *strongly equivalent* to each other if, for any action description D (possibly of a larger signature), $T(D\cup D_1)=T(D\cup D_2)$. This is similar to the definition of strong equivalence for logic programs [46].

Theorem 6.4.1. *Any two static constraints (6.6) with the same atoms $A_1, \dots, A_m$ are strongly equivalent to each other.*

The rules contributed to $PN_l(D)$ by (6.6) can be equivalently written as

$$\bot \leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m.$$

A *dynamic constraint* is a pair of dynamic laws of the form

$$\begin{array}{l} f\!=\!v \textbf{ after } a_1, \dots, a_k, A_1, \dots, A_m \\ f\!=\!w \textbf{ after } a_1, \dots, a_k, A_1, \dots, A_m \end{array} \tag{6.7}$$

where $v \neq w$, $a_1, \dots, a_k$ $(k > 0)$ are action constants, and $A_1, \dots, A_m$ are atoms. We will write (6.7) as

$$\textbf{nonexecutable } a_1, \dots, a_k \textbf{ if } A_1, \dots, A_m,$$

and we will drop **if** in this abbreviation when $m = 0$. The use of this abbreviation depends on the following fact:

Theorem 6.4.2. *Any two dynamic constraints (6.7) with the same action constants $a_1, \dots, a_k$ and the same atoms $A_1, \dots, A_m$ are strongly equivalent to each other.*

The rules contributed to $PN_l(D)$ by (6.7) can be equivalently written as

$$\bot \leftarrow i\!:\!a_1 \wedge \cdots \wedge i\!:\!a_k, i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m.$$

6.5 Example: The Blocks World

We will use $\mathcal{BC}$ to describe the blocks world (Example 2.2.3). We further require that every block should belong to a tower that rests on the table; there are no blocks or groups of blocks "floating in the air."

Let *Blocks* be a finite non-empty set of symbols (block names) that does not include the symbol *Table*. The action description below uses the following fluent and action constants:

- for each $B \in \mathit{Blocks}$, regular fluent constant $\mathit{Loc}(B)$ with domain $\mathit{Blocks} \cup \{\mathit{Table}\}$, and statically determined Boolean fluent constant $\mathit{InTower}(B)$;
- for each $B \in \mathit{Blocks}$ and each $L \in \mathit{Blocks} \cup \{\mathit{Table}\}$, action constant $\mathit{Move}(B, L)$.

In the list of static and dynamic laws, B, B_1 and B_2 are arbitrary elements of *Blocks*, and L is an arbitrary element of $\mathit{Blocks} \cup \{\mathit{Table}\}$. Two different blocks cannot rest on the same block:

$$\textbf{impossible } \mathit{Loc}(B_1){=}B, \mathit{Loc}(B_2) = B \qquad (B_1 \neq B_2).$$

The definition of $\mathit{InTower}(B)$:

$$\begin{array}{l}
\mathit{InTower}(B) \textbf{ if } \mathit{Loc}(B){=}\mathit{Table}, \\
\mathit{InTower}(B) \textbf{ if } \mathit{Loc}(B){=}B_1, \mathit{InTower}(B_1), \\
\textbf{default } \sim\!\mathit{InTower}(B).
\end{array}$$

Blocks don't float in the air:

$$\textbf{impossible } \sim\!\mathit{InTower}(B).$$

The commonsense law of inertia:

$$\textbf{inertial } \mathit{Loc}(B).$$

The effect of moving a block:

$$\mathit{Move}(B,L) \textbf{ causes } \mathit{Loc}(B)=L.$$

A block cannot be moved unless it is clear:

$$\textbf{nonexecutable } \mathit{Move}(B,L) \textbf{ if } \mathit{Loc}(B_1)=B.$$

Figure 6.1 is a representation of logic programs $PN_l(D)$ (Section 6.3), for this action description D, in the input language of the grounder GRINGO.

The values of the symbolic constants `l` (the number of steps) and `n` (the number of blocks) are supposed to be specified in command line. The stable models generated by an answer set solver for this input file will represent all trajectories of length `l` in the transition system corresponding to the blocks world with `n` blocks. For instance, if we ground this program with the GRINGO options

```
-c l=0 -c n=3
```

then the resulting program will have 13 stable models, corresponding to the states of the transition system—to all possible configurations of 3 blocks.

The rules involving `intower` can be written more economically if we use strong negation and replace

```
% declarations of variables for steps, blocks, and locations
step(0..l).
#domain step(I).
block(b(1..n)).
#domain block(B).
#domain block(B1).
#domain block(B2).
location(X) :- block(X).
location(table).
#domain location(L).

% translations of static laws
:- loc(B1,B,I), loc(B2,B,I), B1!=B2.
intower(B,true,I) :- loc(B,table,I).
intower(B,true,I) :- loc(B,B1,I), intower(B1,true,I).
intower(B,false,I).
:- intower(B,false,I).

% translations of dynamic laws
loc(B,L,I+1) :- loc(B,L,I), I<l.
loc(B,L,I+1) :- move(B,L,I), I<l.
:- move(B,L,I), loc(B1,B,I), I<l.

loc(B,L,0).
move(B,L,I) :- I<l.

:- loc(B,LL,I) : location(LL) 0.
:- intower(B,false,I), intower(B,true,I) 0.

:- 2 loc(B,LL,I): location(LL).
:- intower(B,false,I), intower(B,true,I).
```

Figure 6.1: Blocks World in the language of GRINGO

```
intower(B,true,I), intower(B,false,I)
```

with

```
intower(B,I), -intower(B,I).
```

That would make the uniqueness of value constraint for `intower` redundant.

6.6 Example: A Leaking Container

The leaking container (Example 2.2.4) can be described using the regular fluent constants *Amt* with domain $\{0, \dots, n\}$, for the amount of liquid in the container, and the action constant *FillUp*. There are two dynamic laws:

$$\begin{array}{l} \textbf{default } \mathit{Amt}\!=\!\max(a-k,0) \textbf{ after } \mathit{Amt}\!=\!a \;\; (a = 0,\dots,n), \\ \mathit{FillUp} \textbf{ causes } \mathit{Amt}\!=\!n. \end{array}$$

(When $k = 0$, the first of them turns into **inertial** *Amt*.)

Consider the following temporal projection problem involving this domain, with $n = 2$ and $k = 3$: initially the container is full, and it is filled up at time 3; we would like to know how the amount of liquid in the container will change with time.

The program shown in Figure 6.2 consists of the rules of $PN_l(D)$ and rules encoding the temporal projection problem. The solver CLINGO produces the following output:

```
amt(10,0) amt(10,4) amt(7,5) amt(7,1) amt(4,2) amt(4,6)
amt(1,7) amt(1,3) amt(0,9) amt(0,8)
```

```
% declarations of variables for steps and amounts
step(0..l).
#domain step(I).
amount(0..n).
#domain amount(A).

% translations of dynamic laws
amt(AA,I+1) :- amt(A,I), AA=(|A-k|+(A-k))/2, I<l.
amt(n,I+1) :- fillup(I), I<l.

% standard choice rules
amt(A,0).
fillup(I) :- I<l.

% existence of value constraints
:- amt(AA,I) : amount(AA) 0.

% uniqueness of value constraints
:- 2 amt(AA,I) : amount(AA).

% temporal projection
amt(n,0).
fillup(3).  -fillup(0..2;4..l).

#hide.
#show amt/2.
```

Figure 6.2: Leaking Container in the language of GRINGO

6.7 Translation into the Language of Logic Programs with Strong Negation

In the definition of the semantics of $\mathcal{BC}$ in Section 6.3 the programs $PN_l(D)$ can be replaced by the programs with strong negation $PS_l(D)$ that consist of the following rules:

- the translations

$$i\!:\!A_0 \leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \wedge \neg \sim\; i\!:\!A_{m+1} \wedge \cdots \wedge \neg \sim i\!:\!A_n$$

  $(i \leq l)$ of all static laws (6.1) from D,

- the translations

$$(i+1)\!:\!A_0 \leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \wedge \neg \sim (i+1)\!:\!A_{m+1} \wedge \cdots \wedge \neg \sim (i+1)\!:\!A_n$$

  $(i < l)$ of all dynamic laws (6.2) from D,

- the disjunctive rules

$$0\!:\!A \vee \sim 0\!:\!A$$

  for every atom A containing a regular fluent constant,

- the disjunctive rules

$$i\!:\!a \vee \sim i\!:\!a$$

  for every action constant a and every $i < l$,

- the existence of value constraint

$$\leftarrow \neg\, i\!:\!(f\!=\!v_1) \wedge \cdots \wedge : \neg\, i\!:\!(f\!=\!v_k)$$

  for every fluent constant f and every $i \leq l$, where $v_1, \ldots, v_k$ are all elements of the domain of f,

- the uniqueness of value rule

$$\sim i\!:\!(f\!=\!v) \leftarrow i\!:\!(f\!=\!w)$$

for every fluent constant f, every pair of distinct elements v, w of its domain, and every $i \leq l$.

The stable models of the program $PN_l(D)$ from Section 6.3 can be obtained from the (complete) answer sets of $PS_l(D)$ by removing all negative literals:

Theorem 6.7.1. *A set X of atoms of the signature $\sigma_{D,l}$ is a stable model of $PN_l(D)$ iff $X \cup \{\sim A \mid A \in \sigma_{D,l} \setminus X\}$ is an answer set of $PS_l(D)$.*

It follows that the translation *PN* in the definition of $T(D)$ can be replaced with the translation *PS*.

6.8 Relation to $\mathcal{B}$

The version of the action language $\mathcal{B}$ referred to in this section is defined in [31]. For any action description D in the language $\mathcal{B}$, by $D^{\neg}_{\sim}$ we denote the result of replacing each negative literal $\neg f$ in D with the atom $\sim f$ (that is, $f = \mathbf{f}$). The abbreviations introduced in Sections 6.1 and 6.4 above allow us to view $D^{\neg}_{\sim}$ as an action description in the sense of $\mathcal{BC}$, provided that all fluent constants are treated as regular Boolean. We define the translation of D into $\mathcal{BC}$ as the result of extending $D^{\neg}_{\sim}$ by adding the inertiality assumptions (6.5) for all fluent constants f.

We will loosely refer to states and transitions of the transition system represented by D as states and transitions of D.

To state the claim that this translation preserves the meaning of D, we need to relate states and transitions in the sense of the semantics of $\mathcal{B}$ to states and transitions in the sense of Section 6.3. In $\mathcal{B}$, a state is a consistent and complete set of literals f, $\neg f$ for fluent constants f. For any set s of atoms f, $\sim\! f$, by $s^{\sim}_{\neg}$ we denote the set of literals obtained from s by replacing each atom $\sim\! f$ with the negative literal $\neg f$. Furthermore, an action in $\mathcal{B}$ is a consistent and complete set of literals a, $\neg a$ for action constants a.

Theorem 6.8.1. *For any action description D in the language $\mathcal{B}$,*

(a) a set s of atoms is a state of the translation of D into the language $\mathcal{BC}$ iff $s^{\sim}_{\neg}$ is a state of D;

(b) for any sets s_0, s_1 of atoms and any set α of action constants, $\langle s_0, \alpha, s_1\rangle$ is a transition of the translation of D into the language $\mathcal{BC}$ iff

$$\langle (s_0)^{\sim}_{\neg}, \alpha \cup \{\neg 0 : a \mid a \notin \alpha\}, (s_1)^{\sim}_{\neg}\rangle$$

is a transition of D.

The description of the blocks world from Section 6.5 does not correspond to any $\mathcal{B}$-description, in the sense of this translation, for two reasons. First, some fluent constants in it are not regular: it uses statically determined fluents $\mathit{InTower}(B)$, defined recursively in terms of $\mathit{Loc}(B)$. They are similar to "defined fluents" allowed in the extension of $\mathcal{B}$ introduced in [26]. Second, some fluent constants in it are not Boolean: the values of $\mathit{Loc}(B)$ are locations.

The leaking container example (Section 6.6) does not correspond to any $\mathcal{B}$-description either: the regular fluent *Amt* is not Boolean, and the default describing how the value of this fluent changes is different from the commonsense law of inertia. An alternative approach to describing the leaking container is based on an extension of $\mathcal{B}$ by "process fluents," called $\mathcal{H}$ [10].

6.9 Relation to $\mathcal{C}+$

The semantics of $\mathcal{C}+$ is based on the idea of universal causation [59]. Formal relationships between universal causation and stable models are investigated in [18, 58], and it is not surprising that a large fragment of $\mathcal{BC}$ is equivalent to a large fragment of $\mathcal{C}+$.

In $\mathcal{C}+$, just as in $\mathcal{BC}$, some fluent symbols can be designated as "statically determined." (Other fluents are called "simple" in $\mathcal{C}+$; they correspond to regular fluents in our terminology.) Fluent symbols in $\mathcal{C}+$ may be non-exogenous; in $\mathcal{BC}$ such fluents are not allowed. Action symbols in $\mathcal{C}+$ may be non-Boolean; in this respect, that language is more general than the version of $\mathcal{BC}$ defined above.

Consider a $\mathcal{BC}$-description such that, in each of its static laws (6.1), $m = 0$. In other words, we assume that every static law has the form

$$A_0 \textbf{ ifcons } A_1, \ldots, A_n. \tag{6.8}$$

Such a description can be translated into $\mathcal{C}+$ as follows:

- all action constants are treated as Boolean;

- every static law (6.8) is replaced with

$$\textbf{caused } A_0 \textbf{ if } A_1 \wedge \cdots \wedge A_n;$$

- every dynamic law (6.2) is replaced with

$$\textbf{caused } A_0 \textbf{ if } A_{m+1} \wedge \cdots \wedge A_n \textbf{ after } A_1 \wedge \cdots \wedge A_m;$$

- for every action constant a,

$$\textbf{exogenous } a$$

  is added.

Theorem 6.9.1. *For any action description D in the language $\mathcal{BC}$ such that in each of its static laws (6.1) $m = 0$,*

(a) the states of the translation of D into the language $\mathcal{C}+$ are identical to the states of D;

(b) the transitions of the translation of D into the language $\mathcal{C}+$ can be characterized as the triples

$$\langle\, s_0,\ \{a{=}\mathbf{t} \mid a \in \alpha\} \cup \{a{=}\mathbf{f} \mid a \in \sigma^A \setminus \alpha\},\ s_1 \,\rangle$$

for all transitions $\langle s_0, \alpha, s_1\rangle$ of D.

This translation is applicable, for instance, to the leaking container example. The description of the blocks world from Section 6.5 cannot be translated into $\mathcal{C}+$ in this way, because the static laws in the recursive definition of $InTower(B)$ violate the condition $m = 0$.

To sum up, in this chapter we propose a new action language by combining attractive features from both language $\mathcal{B}$ and $\mathcal{C}+$, leading to language $\mathcal{BC}$. The semantics of the action description in $\mathcal{BC}$ is defined by translating into logic program under stable model semantics. The new language can be used to automate reasoning about actions by calling answer set solvers.

Chapter 7

Proofs

Recall that the translations of static laws (6.1) introduced in Section 6.3 are the formulas

$$i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \wedge \neg\neg\, i\!:\!A_{m+1} \wedge \cdots \wedge \neg\neg\, i\!:\!A_n \rightarrow i\!:\!A_0, \tag{7.1}$$

and the translations of dynamic laws (6.2) are

$$i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \wedge \neg\neg\,(i+1)\!:\!A_{m+1} \wedge \cdots \wedge \neg\neg\,(i+1)\!:\!A_n \rightarrow (i+1)\!:\!A_0. \tag{7.2}$$

Using the notation introduced in Section 2.4.3, the choice rules from $PN_l(D)$ can be written as $Choice(0:A)$ and $Choice(i:a)$. Finally, the existence of value and uniqueness of value constraints from Section 6.3 are

$$\neg(\neg\, i\!:\!(f\!=\!v_1) \wedge \cdots \wedge \neg\, i\!:\!(f\!=\!v_k)) \tag{7.3}$$

and

$$\neg(i\!:\!(f\!=\!v) \wedge i\!:\!(f\!=\!w)). \tag{7.4}$$

Formulas of the form (7.1) corresponding to the same static law can be obtained from each other by adding the same number to all time stamps, or by subtracting the same number. We will denote by $F \uparrow i$ the formula obtained from a formula F by adding i to each time stamp; $F\!\downarrow\! i$ is the result of subtracting i from each time stamp. If F is a formula of the form (7.1) with $i = 0$ then the other formulas (7.1) corresponding to the same static law can be written as $F \uparrow 1, \dots, F \uparrow l$. This notation will be used also in connection with formulas (7.2)–(7.4).

Let σ^F be the set of expressions of the form $f\!=\!v$, where f is a fluent constant and v is an element of its domain. Let σ^R be the part of σ^F consisting of the expressions $f\!=\!v$ with regular f, and σ^{SD} be the part of σ^F consisting of expressions in which f is statically determined. By σ^A we denote the set of all action constants.

Let SL be the conjunction of formulas (7.1) with $i = 0$ for all static laws from D. Let DL be the conjunction of formulas (7.2) with $i = 0$ for all dynamic laws from D. Let C be the conjunction of formulas (7.3) and (7.4) with $i = 0$ for all fluent constants f and all pairs of distinct v,w. Then $PN_0(D)$ is

$$SL \wedge \mathit{Choice}(0\!:\!\sigma^R) \wedge C,$$

and $PN_1(D)$ is

$$SL \wedge SL\!\uparrow\! 1 \wedge DL \wedge \mathit{Choice}(0\!:\!\sigma^R, 0\!:\!\sigma^A) \wedge C \wedge C\!\uparrow\! 1.$$

Here is how Facts 2.4.1(b) and 2.4.2(b) can be used to characterize states and transitions of an action description. An interpretation s_0 of the signature $\sigma^R \cup \sigma^{SD}$ is a state iff $0\!:\!s_0$ is a stable model of PN_0, that is,

$$0\!:\!s_0 \models \mathrm{SM}[SL \wedge \mathit{Choice}(0\!:\!\sigma^R) \wedge C; 0\!:\!\sigma^F].$$

This condition is equivalent to

$$0\!:\!s_0 \models \mathrm{SM}[SL; 0\!:\!\sigma^{SD}] \wedge C. \tag{7.5}$$

For any interpretations s_0 and s_1 of the signature σ^F and any interpretation α of the signature σ^A, $\langle 0\!:\!s_0, 0\!:\!\alpha, 1\!:\!s_1\rangle$ is a transition iff $0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1$ is a stable model of $PN_1(D)$, that is, iff

$$\begin{aligned} 0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1 \models \mathrm{SM}[SL \wedge SL{\uparrow}1 \wedge DL \wedge \mathit{Choice}(0\!:\!\sigma^R, 0\!:\!\sigma^A) \wedge C \wedge C{\uparrow}1;\\ 0\!:\!\sigma^F \cup 0\!:\!\sigma^A \cup 1\!:\!\sigma^F].\end{aligned}$$

This condition is equivalent to

$$0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1 \models \mathrm{SM}[SL \wedge SL{\uparrow}1 \wedge DL; 0\!:\!\sigma^{SD} \cup 1\!:\!\sigma^F] \wedge C \wedge C{\uparrow}1. \tag{7.6}$$

Furthermore, by Fact 2.4.4, that the conjunctive term

$$\mathrm{SM}[SL \wedge SL{\uparrow}1 \wedge DL; 0\!:\!\sigma^{SD} \cup 1\!:\!\sigma^F]$$

from (7.6) is equivalent to

$$\mathrm{SM}[SL; 0\!:\!\sigma^{SD}] \wedge \mathrm{SM}[SL{\uparrow}1 \wedge DL; 1\!:\!\sigma^F].$$

Consequently, σ^A, $\langle 0\!:\!s_0, 0\!:\!\alpha, 1\!:\!s_1\rangle$ is a transition iff

$$0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1 \models \mathrm{SM}[SL; 0\!:\!\sigma^{SD}] \wedge \mathrm{SM}[SL{\uparrow}1 \wedge DL; 1\!:\!\sigma^F] \wedge C \wedge C{\uparrow}1. \tag{7.7}$$

7.1 Proof of Theorem 6.3.1

Theorem 6.3.1. *For every transition $\langle s_0, \alpha, s_1\rangle$, s_0 and s_1 are states.*

Proof. Condition (7.5), expressing that s_0 is a state, immediately follows from (7.7). The assertion that s_1 is a state is expressed by the condition

$$0\!:\!s_1 \models \mathrm{SM}[SL; 0\!:\!\sigma^{SD}] \wedge C,$$

or, equivalently,

$$1\!:\!s_1 \models \mathrm{SM}[SL{\uparrow}1; 1\!:\!\sigma^{SD}] \wedge C{\uparrow}1. \tag{7.8}$$

Assume (7.7). Recall that DL is the conjunction of implications (7.2) with $i = 0$. By DL' we denote the formula obtained from DL by replacing each of the conjunctive terms $0\!:\!A_j$ $(1 \leq j \leq m)$ in the antecedents of these implications with $\top$ if $A_j \in s_0 \cup \alpha$, and with $\bot$ otherwise. Let α^* is the union of α with the set of literals $\neg a$ for all action constants a that do not belong to α. The equivalence between each of the atoms $0\!:\!A_j$ from DL and the truth value that replaced it in DL' is an intuitionistic consequence of the formulas

$$0\!:\!s_0,\ 0\!:\!\alpha^*,\ C. \tag{7.9}$$

Indeed, each atom $0\!:\!A_j$ that is replaced by $\top$ belongs to $0\!:\!s_0 \cup 0\!:\!\alpha^*$. For each atom of the form $0\!:\!a$ that is replaced by $\bot$, its negation belongs to $0\!:\!\alpha^*$. For each atom of the form $0\!:\!(f = v)$ that is replaced by $\bot$, its negation is an intuitionistic consequence of the atom $0\!:\!(f = w)$ from $0\!:\!s_0$, where w is the value of f in state s_0, and the uniqueness constraint

$$\neg(0\!:\!(f = v) \wedge 0\!:\!(f = w))$$

from C.

Thus the equivalence $DL \leftrightarrow DL'$ is an intuitionistic consequence of formulas (7.9), so that the conjunction

$$SL{\uparrow}1 \wedge DL \wedge C \wedge 0\!:\!s_0 \wedge 0\!:\!\alpha^*$$

is intuitionistically equivalent to

$$SL{\uparrow}1 \wedge DL' \wedge C \wedge 0\!:\!s_0 \wedge 0\!:\!\alpha^*.$$

By Fact 2.4.3, it follows that

$$\mathrm{SM}[SL{\uparrow}1 \wedge DL \wedge C \wedge 0\!:\!s_0 \wedge 0\!:\!\alpha^*; 0\!:\!\sigma^{SD} \cup 1\!:\!\sigma^F] \tag{7.10}$$

is equivalent to

$$\mathrm{SM}[SL{\uparrow}1 \wedge DL' \wedge C \wedge 0\!:\!s_0 \wedge 0\!:\!\alpha^*; 0\!:\!\sigma^{SD} \cup 1\!:\!\sigma^F]. \tag{7.11}$$

(In these formulas, $0\!:\!s_0$ is the conjunction of the elements of the set $0\!:\!s_0$, and $0\!:\!\alpha^*$ is understood in a similar way.)

From (7.7), by Fact 2.4.2(b),

$$0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1 \models \mathrm{SM}[SL{\uparrow}1 \wedge DL \wedge C; 0\!:\!\sigma^{SD} \cup 1\!:\!\sigma^F].$$

Since

$$0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1 \models 0\!:\!s_0 \wedge 0\!:\!\alpha^*,$$

we further conclude, by Fact 2.4.2(a), that $0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1$ satisfies (7.10), and consequently (7.11).

By Fact 2.4.4, formula (7.11) can be equivalently rewritten as the conjunction of

$$\mathrm{SM}[SL{\uparrow}1 \wedge DL'; 1{:}\sigma^F] \tag{7.12}$$

and

$$\mathrm{SM}[C \wedge 0{:}s_0 \wedge 0{:}\alpha^*; 0{:}\sigma^{SD}].$$

Hence the interpretation $1 : s_1$ of the signature $1 : \sigma^F$ satisfies (7.12). By Fact 2.4.1(a), it follows that

$$1{:}s_1 \models \mathrm{SM}[SL{\uparrow}1 \wedge DL'; 1{:}\sigma^{SD}].$$

Then, by Fact 2.4.2(b),

$$1{:}s_1 \models \mathrm{SM}[SL{\uparrow}1; 1{:}\sigma^{SD}].$$

Assertion (7.8) follows from this formula and (7.7).

The following remark will be useful in the proof of Theorem 6.3.2: the assertion that s_1 is a state for any transition $\langle s_0, \alpha, s_1\rangle$ can be expressed by saying that the sentence

$$\mathrm{SM}[SL \wedge SL{\uparrow}1 \wedge DL; 0{:}\sigma^{SD} \cup 1{:}\sigma^F] \wedge C \wedge C{\uparrow}1 \tag{7.13}$$

from (7.6) entails the sentence

$$\mathrm{SM}[SL{\uparrow}1; 1{:}\sigma^{SD}] \tag{7.14}$$

from (7.8).

7.2 Proof of Theorem 6.3.2

Theorem 6.3.2. *For every $l \geq 1$, X is a stable model of $PN_l(D)$ iff $X^0, \dots, X^{l-1}$ are transitions.*

Proof. For any set X of elements of the signature $\sigma_{l,D}$, X^i can be described as the triple $\langle s_0, \alpha, s_1 \rangle$ satisfying the conditions

$$\begin{aligned} i\!:\!s_0 &= X \cap (i\!:\!\sigma^F), \\ i\!:\!\alpha &= X \cap (i\!:\!\sigma^A), \\ (i+1)\!:\!s_1 &= X \cap (i+1\!:\!\sigma^F), \end{aligned}$$

or, equivalently,

$$\begin{aligned} 0\!:\!s_0 &= (X \cap (i\!:\!\sigma^F))\!\downarrow\! i, \\ 0\!:\!\alpha &= (X \cap (i\!:\!\sigma^A))\!\downarrow\! i, \\ 1\!:\!s_1 &= (X \cap (i+1\!:\!\sigma^F))\!\downarrow\! i. \end{aligned}$$

The set $0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1$ for this triple can be written as

$$(X \cap (i\!:\!\sigma^F \cup i\!:\!\sigma^A \cup i+1\!:\!\sigma^F))\!\downarrow\! i.$$

In view of the characterization of transitions given by formula (7.6), X^i is a transition iff

$$\begin{aligned} &(X \cap (i\!:\!\sigma^F \cup i\!:\!\sigma^A \cup i+1\!:\!\sigma^F))\!\downarrow\! i \models \\ &\qquad \mathrm{SM}[SL \wedge SL\!\uparrow\!1 \wedge DL; 0\!:\!\sigma^{SD} \cup 1\!:\!\sigma^F] \wedge C \wedge C\!\uparrow\!1, \end{aligned}$$

or, equivalently,

$$X \models \mathrm{SM}[SL\!\uparrow\! i \wedge SL\!\uparrow\!(i+1) \wedge DL\!\uparrow\! i; i\!:\!\sigma^{SD} \cup (i+1)\!:\!\sigma^F] \wedge C\!\uparrow\! i \wedge C\!\uparrow\!(i+1). \quad (7.15)$$

Consequently Theorem 6.3.2 can be expressed by saying that the sentence

$$\mathrm{SM}\left[PN_l(D); \bigcup_{i=0}^{l} i\!:\!\sigma^F \cup \bigcup_{i=0}^{l-1} i\!:\!\sigma^A\right] \quad (7.16)$$

is equivalent to the conjunction

$$\bigwedge_{i=0}^{l-1} \big(\mathrm{SM}[SL{\uparrow}i \wedge SL{\uparrow}(i+1) \wedge DL{\uparrow}i; i\,{:}\,\sigma^{SD} \cup (i+1)\,{:}\,\sigma^F] \wedge C{\uparrow}i \wedge C{\uparrow}(i+1)\big) \tag{7.17}$$

of sentences from (7.15).

Let us show first that (7.17) entails (7.16). By Fact 2.4.4, (7.17) is equivalent to

$$\bigwedge_{i=0}^{l-1} \big(\mathrm{SM}[SL{\uparrow}i; i\,{:}\,\sigma^{SD}] \wedge \mathrm{SM}[SL{\uparrow}(i+1) \wedge DL{\uparrow}i; i+1\,{:}\,\sigma^F]\big) \wedge \bigwedge_{i=0}^{l} C{\uparrow}i,$$

which implies

$$\mathrm{SM}[SL; 0\,{:}\,\sigma^{SD}] \wedge \bigwedge_{i=0}^{l-1} \mathrm{SM}[SL{\uparrow}(i+1) \wedge DL{\uparrow}i; i+1\,{:}\,\sigma^F] \wedge \bigwedge_{i=0}^{l} C{\uparrow}i.$$

By Fact 2.4.4, this formula can be equivalently rewritten as

$$\mathrm{SM}\left[SL \wedge \bigwedge_{i=0}^{l-1}(SL{\uparrow}(i+1) \wedge DL{\uparrow}i); 0\,{:}\,\sigma^{SD} \cup \bigcup_{i=1}^{l} i\,{:}\,\sigma^F\right] \wedge \bigwedge_{i=0}^{l} C{\uparrow}i,$$

which, by Facts 2.4.1(b) and 2.4.2(b), is equivalent to (7.17).

We will now show that (7.16) entails (7.17) by induction on l. If $l = 1$ then (7.16) is as

$$\mathrm{SM}[PN_1(D); 0\,{:}\,\sigma^F \cup 0\,{:}\,\sigma^A \cup 1\,{:}\,\sigma^F],$$

and (7.17) is

$$\mathrm{SM}[SL \wedge SL{\uparrow}1 \wedge DL; 0\,{:}\,\sigma^{SD} \cup 1\,{:}\,\sigma^F] \wedge C \wedge C{\uparrow}1.$$

The latter follows from the former by Facts 2.4.1(b) and 2.4.2(b).

Assume, for some value of l, that (7.16) entails (7.17), and assume

$$\mathrm{SM}\left[PN_{l+1}(D); \bigcup_{i=0}^{l+1} i\!:\!\sigma^F \cup \bigcup_{i=0}^{l} i\!:\!\sigma^A\right]. \tag{7.18}$$

Our goal is to derive

$$\mathrm{SM}[SL{\uparrow}i \wedge SL{\uparrow}(i+1) \wedge DL{\uparrow}i; i\!:\!\sigma^{SD} \cup (i+1)\!:\!\sigma^F] \wedge C{\uparrow}i \wedge C{\uparrow}(i+1). \tag{7.19}$$

for $i = 0, \ldots, l$. First, notice that $PN_{l+1}(D)$ is

$$PN_l(D) \wedge SL{\uparrow}(l+1) \wedge DL{\uparrow}l \wedge \mathit{Choice}(l\!:\!\sigma^A) \wedge C{\uparrow}(l+1).$$

By the splitting theorem (Fact 2.4.4), from (7.18) we conclude

$$\mathrm{SM}\left[PN_l(D); \bigcup_{i=0}^{l} i\!:\!\sigma^F \cup \bigcup_{i=0}^{l-1} i\!:\!\sigma^A\right] \tag{7.20}$$

and

$$\mathrm{SM}[SL{\uparrow}(l+1) \wedge DL{\uparrow}l \wedge \mathit{Choice}(l\!:\!\sigma^A) \wedge C{\uparrow}(l+1); (l+1)\!:\!\sigma^F \cup l\!:\!\sigma^A]. \tag{7.21}$$

By the induction hypothesis, (7.20) implies (7.19) for $i = 0, \ldots, l-1$. It remains to derive (7.19) for $i = l$.

We have already proved (7.19) for $i = l-1$, that is,

$$\mathrm{SM}[SL{\uparrow}(l-1) \wedge SL{\uparrow}l \wedge DL{\uparrow}(l-1); (l-1)\!:\!\sigma^{SD} \cup l\!:\!\sigma^F] \wedge C{\uparrow}(l-1) \wedge C{\uparrow}l. \tag{7.22}$$

As we observed in Section 7.1, formula (7.13) entails (7.14). Consequently (7.22) entails

$$\mathrm{SM}[SL{\uparrow}l; l\!:\, \sigma^{SD}].$$

Using the last term of (7.22), we can further conclude, using Fact 2.4.2(b), that

$$\mathrm{SM}[SL{\uparrow}l \wedge C{\uparrow}l; l\colon \sigma^{SD}].$$

On the other hand, from (7.21) we conclude by Fact 2.4.1(b) that

$$\mathrm{SM}[SL{\uparrow}(l+1) \wedge DL{\uparrow}l \wedge C{\uparrow}(l+1); (l+1)\colon \sigma^{F}].$$

By Fact 2.4.4, it follows that

$$\mathrm{SM}[SL{\uparrow}l \wedge C{\uparrow}l \wedge SL{\uparrow}(l+1) \wedge DL{\uparrow}l \wedge C{\uparrow}(l+1); l\colon \sigma^{SD} \cup (l+1)\colon \sigma^{F}],$$

which is (7.19) for $i = l$.

7.3 Proofs of Theorems 6.4.1 and 6.4.2

Theorem 6.4.1. *Any two static constraints (6.6) with the same atoms $A_1, \ldots, A_m$ are strongly equivalent to each other.*

Proof. We need to show that for any action description D and any two static constraints SC_1

$$\begin{array}{c} f = v \text{ if } A_1, \ldots, A_m \\ f = w \text{ if } A_1, \ldots, A_m \end{array}$$

and SC_2

$$\begin{array}{c} f' = v' \text{ if } A_1, \ldots, A_m \\ f' = w' \text{ if } A_1, \ldots, A_m, \end{array}$$

$T(D \cup SC_1) = T(D \cup SC_2)$. It is sufficient to check that the logic programs $PN_l(D \cup SC_1)$ and $PN_l(D \cup SC_2)$, which describe the states and transitions of $T(D \cup SC_1)$ and $T(D \cup SC_2)$ when $l = 0, 1$, have the same stable models. We will show that these programs are intuitionistically equivalent to each other, and the claim that they have the same stable models will follow by Fact 2.4.3.

Program $PN_l(D \cup SC_1)$ is the conjunction of $PN_l(D)$ with the implications

$$\begin{array}{c} i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \to i\!:\!(f = v), \\ i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m \to i\!:\!(f = w) \end{array}$$

$(i = 0, \ldots, l)$. Since $PN_l(D)$ contains the formulas

$$\neg(i\!:\!(f = v) \wedge i\!:\!(f = w)),$$

$PN_l(D \cup SC_1)$ is intuitionistically equivalent to

$$PN_l(D) \wedge \bigwedge_{i=0}^{l} \neg(i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_m). \tag{7.23}$$

Similarly, $PN_1(D \cup SC_2)$ is intuitionistically equivalent to (7.23) as well.

The proof of Theorem 6.4.2 is similar.

7.4 Proof of Theorem 6.7.1

Consider the logic program $PN'_l(D)$ obtained from $PN_l(D)$ by

(a) adding to its signature a new atom $\overline{A}$ for every atom A,

(b) adding the rule

$$\overline{A} \leftarrow \neg\, A \tag{7.24}$$

for every A,

(c) adding the constraint

$$\leftarrow A, \overline{A} \tag{7.25}$$

for every A, and

(d) replacing the uniqueness of value constraints

$$\leftarrow i\!:\!(f\!=\!v), i\!:\!(f\!=\!w) \tag{7.26}$$

by the rules

$$\overline{i\!:\!(f = v)} \leftarrow i\!:\!(f = w). \tag{7.27}$$

Lemma 7.4.1. *X is a stable model of $PN_l(D)$ iff $X \cup \{\overline{A} : A \in \sigma_{l,D} \setminus X\}$ is a stable model of $PN'_l(D)$.*

Proof. Let X' denote $X \cup \{\overline{A} : A \in \sigma_{l,D} \setminus X\}$, and the program obtained from $PN_l(D)$ after steps (a)–(c) as $PN''_l(D)$.

First, by Lemma on Explicit Definitions from [16], for the program obtained from $PN_l(D)$ after steps (a) and (b), X is a stable model of $PN_l(D)$ iff X' is its stable model. Second, since $\overline{A} \in X'$ iff $A \notin X'$, X' satisfies the constraints (7.25), and therefore X' is a stable model of $PN''_l(D)$. To prove X' is a stable model of $PN'_l(D)$, by Fact 2.4.3, it remains to prove that $PN''_l(D)$ is intuitionistically equivalent to the program $PN'_l(D)$. It is easy to see that the set of formulas (7.24) and (7.26) is intuitionistically equivalent to (7.24), (7.25) and (7.27).

Theorem 6.7.1 *A set X of atoms of the signature $\sigma_{l,D}$ is a stable model of $PN_l(D)$ iff $X \cup \{\sim A : A \in \sigma_{l,D} \setminus X\}$ is an answer set of $PS_l(D)$.*

Proof. As observed in Section 4.1.3, a negative atom can be replaced in a logic program by a regular atom if we add a constraint that does not allow both atoms to be in the same stable model. In this spirit, for each atom A from $\sigma_{l,D}$, we choose a new atom $\overline{A}$ which will be used in place of the negative atom $\sim A$. Let $PS_l(D)^+$ be the program obtained from $PS_l(D)$ by replacing each negative atom $\sim A$ with $\overline{A}$ and adding the rules

$$\leftarrow A, \overline{A} \tag{7.28}$$

for all atoms A. For any set X of atoms of the signature $\sigma_{l,D}$,

$$X \cup \{\sim A \,:\, A \in \sigma_{l,D} \setminus X\} \text{ is an answer set of } PS_l(D) \tag{7.29}$$

iff

$$X \cup \{\overline{A} \,:\, A \in \sigma_{l,D} \setminus X\} \text{ is a stable model of } PS_l(D)^+. \tag{7.30}$$

Consider now the program obtained from $PS_l(D)^+$ by

(a) replacing the rules

$$0\!:\!(f\!=\!v) \vee \overline{0\!:\!(f\!=\!v)} \tag{7.31}$$

for regular fluent constants f with

$$\begin{array}{l} 0\!:\!(f\!=\!v) \leftarrow \neg\, \overline{0\!:\!(f\!=\!v)}, \\ \overline{0\!:\!(f\!=\!v)} \leftarrow \neg\, 0\!:\!(f\!=\!v), \end{array} \tag{7.32}$$

(b) replacing the rules

$$i\!:\!a \vee \overline{i\!:\!a} \tag{7.33}$$

for action constants a with

$$\begin{array}{l} i\!:\!a \leftarrow \neg\, \overline{i\!:\!a}, \\ \overline{i\!:\!a} \leftarrow \neg\, i\!:\!a, \end{array} \tag{7.34}$$

and

(c) replacing all expressions of the form $\neg\overline{A}$ with $\neg\neg A$.

These transformations do not affect the stable models of the program. For steps (a) and (b), this assertion follows from the results on strong equivalence proved in [46], because the program contains constraints (7.28); see the discussion of formulas (4) and (5) in that paper. For step (c), it is sufficient to check, in view of Fact 2.4.3, that this is an intuitionistically equivalent transformation. In the syntax of propositional formulas, step (c) consists in replacing $\neg\overline{A}$ with $\neg\neg A$; we need to show that the equivalence between these two formulas intuitionistically follows from the program obtained after steps (a) and (b). By the Glivenko theorem [35], it is sufficient to derive $\neg\overline{A} \leftrightarrow A$ from this program in classical propositional logic.

Indeed, some of the rules of the program obtained after steps (a) and (b), written as propositional formulas, are

$$\neg(\neg i\!:\!(f = v_1) \wedge \cdots \neg \vee\, i\!:\!(f = v_k)) \tag{7.35}$$

for every fluent f where $v_1, \ldots, v_k$ are all elements of the domain of f (existence of value rules),

$$i\!:\!(f = w) \rightarrow \overline{i\!:\!(f = v)} \tag{7.36}$$

for each fluent f and all $v, w \in Dom(f)$ such that $v \neq w$ (uniqueness of value rules),

$$\neg(i\!:\!(f\!=\!v) \wedge \overline{i\!:\!(f\!=\!v)}) \tag{7.37}$$

for every fluent constant f and all $v \in Dom(f)$ (rule (7.28) with $i:(f=v)$ as A),

$$\neg(i\!:\!a \wedge \overline{i\!:\!a}) \tag{7.38}$$

for every action constant a (rule (7.28) with $i\!:\!a$ as A), and

$$\neg\ \overline{i\!:\!a} \rightarrow i\!:\!a \tag{7.39}$$

for every action constant a (rule (7.34) with $i\!:\!a$ as A).

It is easy to see that

$$i\!:\!a \leftrightarrow \neg\ i\!:\!\overline{a}$$

follows from (7.38) and (7.39), for action constant a. For an atom of the form $i\!:\!(f = v)$, we need to derive the equivalence

$$i\!:\!(f = v) \leftrightarrow \neg\ \overline{i\!:\!(f = v)}$$

First, the implication left-to-right follows from (7.37). To prove the implication right-to-left, assuming $\neg\ \overline{i\!:\!(f = v)}$, by (7.36), we derive $\neg i:(f = w)$ for all $w \in Dom(f) \setminus \{v\}$, and $f = v$ follows from (7.35).

It remains to observe that the obtained program is $PN'_l(D)$. Therefore, (7.30) is equivalent to saying that X is a stable model of $PN_l(D)$.

7.5 Proof of Theorem 6.8.1

In the following proof, we refer to the definition of the semantics of $\mathcal{B}$ given in [31]. In that paper, a consistent and complete set X of propositional literals is identified with the propositional interpretation that satisfies X. Note that this is different from the convention adopted in this paper: we identify that interpretation with the set X^+ of atoms that belong to X.

Theorem 6.8.1. *For any action description D in the language $\mathcal{B}$,*

(a) the states of the translation of D into the language $\mathcal{BC}$ can be characterized as the sets $s^{\neg}_{\sim}$ for all states s of D;

(b) the transitions of the translation of D into the language $\mathcal{BC}$ can be characterized as the triples

$$\langle\, (s_0)^{\neg}_{\sim},\, \alpha \cap \sigma^A,\, (s_1)^{\neg}_{\sim} \,\rangle$$

for all transitions $\langle s_0, \alpha, s_1 \rangle$ of D.

Proof. Take an action description D in the language $\mathcal{B}$. To prove part (a), we need to show that for any consistent and complete set s of fluent literals, $s^{\neg}_{\sim}$ is a state of the translation of D into the language $\mathcal{BC}$ iff s is a state of D, that is to say, iff s^+ satisfies the formulas

$$\neg(L_1 \wedge \cdots \wedge L_m \wedge \neg L_0) \tag{7.40}$$

for all static laws

$$L_0 \textbf{ if } L_1, \ldots, L_m \tag{7.41}$$

of D.

The set $s^{\neg}_{\sim}$ is a state of the translation of D into the language $\mathcal{BC}$ iff $0\!:\!s^{\neg}_{\sim}$ is a stable model of the program $PN_0(D^{\neg}_{\sim})$. This program consists of the rules

$$0\!:\!(L_0)^{\neg}_{\sim} \leftarrow 0\!:\!(L_1)^{\neg}_{\sim}, \dots, 0\!:\!(L_m)^{\neg}_{\sim}$$

for all static laws (7.41) of D, and the choice rules

$$\{0\!:\!f\},\ \{0\!:\!\sim\! f\}$$

and the constraints

$$\begin{array}{l} \leftarrow 0\!:\!f, 0\!:\!\sim\! f, \\ \leftarrow \neg\, 0\!:\!f, \neg\, 0\!:\!\sim\! f \end{array}$$

for all fluent constants f. These rules, rewritten as formulas, can be converted by intuitionistically equivalent transformations into the formulas

$$\neg(0\!:\!(L_1)^{\neg}_{\sim} \wedge \cdots \wedge 0\!:\!(L_m)^{\neg}_{\sim} \wedge \neg 0\!:\!(L_0)^{\neg}_{\sim}) \tag{7.42}$$

for all static laws (7.41) of D, and

$$0\!:\!f \vee \neg 0\!:\!f,\ 0\!:\!\sim\! f \vee \neg 0\!:\!\sim\! f, \tag{7.43}$$

$$\neg(0\!:\!f \wedge 0\!:\!\sim\! f),\ \neg(\neg\, 0\!:\!f \wedge \neg\, 0\!:\!\sim\! f) \tag{7.44}$$

for all fluent constants f. In view of Fact 2.4.2(b), the set $0\!:\!s^{\neg}_{\sim}$ is a stable model of these formulas iff it satisfies all formulas (7.42) and (7.44), or, equivalently, iff $s^{\neg}_{\sim}$ satisfies the formulas

$$\sim\! f \leftrightarrow \neg f \tag{7.45}$$

and (7.40). It remains to observe that $s^{\neg}_{\sim}$ satisfies (7.45), because s is consistent and complete, and that every atom occurring in (7.40) belongs to $s^{\neg}_{\sim}$ iff it belongs to s^{+}.

To prove (b), according to [31], a $\mathcal{B}$ action description D can be translated into *its logic programming representation* LP(D) that consists of rules $S_{\mathcal{B}}$

$$i\!:\!L \leftarrow i\!:\!L_1, \dots i\!:\!L_m \quad (i = 0, 1),$$

for each static law

$$L \textbf{ if } L_1, \dots L_m$$

where $L_1, \dots L_m$ are fluent literals; rules $D_{\mathcal{B}}$

$$1\!:\!L \leftarrow 0\!:\!a, 0\!:\!L_1, \dots 0\!:\!L_n$$

for each dynamic law

$$a \textbf{ causes } L \textbf{ if } L_1, \dots L_n$$

where $L, L_1, \dots L_n$ are fluent literal an a is an elementary action; rules $IN_{\mathcal{B}}$

$$\begin{aligned} 1\!:\!f &\leftarrow 0\!:\!f, \neg \sim 1\!:\!f \\ \sim 1\!:\!f &\leftarrow \sim 0\!:\!f, \neg\, 1\!:\!f \end{aligned}$$

for each fluent constant f.

$C_{\mathcal{B}}$ expresses that the rules

$$\begin{aligned} 0\!:\!f &\leftarrow \neg \sim 0\!:\!f \\ \sim 0\!:\!f &\leftarrow \neg 0\!:\!f \end{aligned}$$

for each fluent constant f, and

$$\begin{aligned} 0\!:\!a &\leftarrow \neg \sim 0\!:\!a \\ \sim 0\!:\!a &\leftarrow \neg\, 0\!:\!a \end{aligned}$$

for all elementary actions a. By [31, Lemma 2], for any sets s_0, s_1 of fluent literals, and any action α, $\langle s_0, \alpha, s_1\rangle$ is a transition of $T(D)$ iff the set

$$\{0\!:\!l \mid l \in s_0 \cup \alpha\} \cup \{1 : l \mid l \in s_1\}$$

is an answer set of the program $\mathrm{LP}(D) \cup C_{\mathcal{B}}$.

Translating a $\mathcal{B}$ action description D into language $\mathcal{BC}$ we obtain $PN_1(D)$ and a set IN' of rules

$$\begin{array}{c} 1\!:\!f \leftarrow 0\!:\!f, \neg\neg\, 1\!:\!f, \\ 1\!:\!\sim\! f \leftarrow 0\!:\!\sim\! f, \neg\neg\, 1\!:\!\sim\! f, \end{array}$$

for each fluent constant f. Therefore, it is sufficient to prove that a set X of literals is an answer set of $\mathrm{LP}(D) \cup C_{\mathcal{B}}$, iff

$$\{i\!:\!f \mid i\!:\!f \in X, i = 0, 1\} \cup \{i\!:\!\sim\! f \mid \neg i\!:\!f \in X, i = 0, 1\} \cup \{0\!:\!a : a \in X\} \tag{7.46}$$

is a stable model of $PN_1(D) \cup IN$', where f is a fluent constant and a is an action constant.

Let $\mathrm{LP}'(D)$, $S'_{\mathcal{B}}$, $D'_{\mathcal{B}}$, $IN'_{\mathcal{B}}$ and $C'_{\mathcal{B}}$ denote the sets of rules obtained by substituting $\sim\! i\!:\!f$ for each literal $\neg i\!:\!f$, and $i\!:\!\widehat{a}$ for each literal $\neg i\!:\!a$ in $\mathrm{LP}(D)$, $S_{\mathcal{B}}$, $D_{\mathcal{B}}$, $IN'_{\mathcal{B}}$ and $C_{\mathcal{B}}$, respectively. By N we denote the set of constraints

$$\leftarrow i\!:\!f, \sim\! i\!:\!f$$

for all fluent constants f, and

$$\leftarrow i\!:\!a, i\!:\!\widehat{a}. \tag{7.47}$$

for all action constants a. Then by [28, Proposition 2],

$$X \text{ is an answer set of } \mathrm{LP}(D) \cup C_{\mathcal{B}} \tag{7.48}$$

iff

$$X' \text{ is a stable model of } \mathrm{LP}'(D) \cup C'_{\mathcal{B}} \cup N \tag{7.49}$$

where X' is

$$\begin{array}{l}\{i\!:\!f \mid i\!:\!f \in X, i = 0, 1\} \cup \{\sim\! i\!:\!f \mid \neg i\!:\!f \in X, i = 0, 1\} \\ \quad \cup \{0\!:\!a \,|\, a \in X\} \cup \{0\!:\!\widehat{a} \,|\, \neg a \in X\}.\end{array}$$

Since a transition of $\mathcal{B}$ is a complete set of literals, for any fluent f,

$$i\!:\!f \in X \text{ or } i\!:\!\neg f \in X, \quad (i = 0, 1)$$

and for elementary action a,

$$0\!:\!a \in X \text{ or } 0\!:\!\neg a \in X,$$

due to $C_{\mathcal{B}}$. Therefore, for the stable model X' of $\mathrm{LP}'(D) \cup C'_{\mathcal{B}} \cup N$, for any fluent f,

$$i\!:\!f \in X' \text{ or } i\!:\!\sim\! f \in X', \quad (i = 0, 1)$$

and for any action constant a,

$$0\!:\!a \in X \text{ or } 0\!:\!\widehat{a} \in X.$$

Therefore, (7.49) iff

$$X' \text{ is a stable model of } \mathrm{LP}'(D) \cup C'_{\mathcal{B}} \cup N \cup \mathit{COMP}. \tag{7.50}$$

where $COMP$ is the set of constraints

$$\leftarrow \neg\ i\!:\!f, \neg\ \sim\!i\!:\!f, \qquad (i = 0, 1)$$

for all fluent constants f, and

$$\leftarrow \neg\ 0\!:\!a, \neg\ 0\!:\!\widehat{a}, \qquad (i = 0, 1) \tag{7.51}$$

for all action constants a.

Furthermore, $N \cup Comp \cup C'_{\mathcal{B}} \cup IN'_{\mathcal{B}}$ is strongly equivalent to the set of rules consisting of $N \cup Comp$, choice rules CH

$$0\!:\!f \vee \neg\ 0\!:\!f, \quad 0\!:\!\sim\!f \vee \neg\ 0\!:\!\sim\!f, \quad 0\!:\!a \vee \neg\ 0\!:\!a$$

and

$$\widehat{0\!:\!a} \leftarrow \neg\ 0\!:\!a \tag{7.52}$$

for all fluent constants f and action constants a, and IN'.

Therefore, $\text{LP}'(D) \cup C'_{\mathcal{B}} \cup N \cup COMP$ in (7.50) is strongly equivalent to

$$S'_{\mathcal{B}} \cup D'_{\mathcal{B}} \cup IN' \cup CH \cup N \cup COMP$$

So (7.50) iff

$$X' \text{ is a stable model of } S'_{\mathcal{B}} \cup D'_{\mathcal{B}} \cup IN' \cup CH \cup N \cup COMP. \tag{7.53}$$

Observe that $S'_{\mathcal{B}} \cup D'_{\mathcal{B}} \cup IN' \cup CH \cup N \cup COMP$ is $PN_1(D) \cup IN'$ union with explicit definitions (7.52), constraints (7.47) and (7.51). The explicit definitions guarantee that the constraints (7.47) and (7.51) are not violated, and therefore they can be dropped. By Lemma of Explicit Definitions [16], we conclude that (7.53) iff (7.46) is a stable model of $PN_1(D) \cup IN'$.

7.6 Proof of Theorem 6.9.1

Theorem 6.9.1 *For any action description D in the language $\mathcal{BC}$ such that in each of its static laws (6.1) $m = 0$,*

(a) the states of the translation of D into the language $\mathcal{C}+$ are identical to the states of D;

(b) the transitions of the translation of D into the language $\mathcal{C}+$ can be characterized as the triples

$$\langle\, s_0,\ \{a{=}\mathbf{t} \,|\, a \in \alpha\} \cup \{a{=}\mathbf{f} \,|\, a \in \sigma^A \setminus \alpha\},\ s_1 \,\rangle \tag{7.54}$$

for all transitions $\langle s_0, \alpha, s_1\rangle$ of D.

Proof. Consider the $\mathcal{C}+$ action description $D_{\mathcal{C}}$ obtained by translating a $\mathcal{BC}$ description D. According to [33, Section 4], a multi-valued interpretation s of fluent constants is a state of $D_{\mathcal{C}}$ iff it is a model of the nonmonotonic causal theory $D^0_{\mathcal{C}}$ that consists of the causal rules

$$0\!:\!A_0 \Leftarrow 0\!:\!A_1 \wedge \cdots \wedge 0\!:\!A_n$$

for the static laws (6.8) of D and the rules

$$0\!:\!f = v \Leftarrow 0\!:\!f = v$$

for all regular fluents. Furthermore, for any multi-valued interpretations s_0, s_1 of fluent constants and any set α of action constants, the triple (7.54) is a transition of $D_{\mathcal{C}}$ iff

$$0\!:\!s_0 \cup \{0\!:\!a\!=\!\mathbf{t} \,|\, a \in \alpha\} \cup \{0\!:\!a\!=\!\mathbf{f} \,|\, a \in \sigma^A \setminus \alpha\} \cup 1\!:\!s_1 \tag{7.55}$$

is a model of the nonmonotonic causal theory $D^1_{\mathcal{C}}$ that consists of the causal rules

$$i\!:\!A_0 \Leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_n \quad (i = 0, 1)$$

for the static laws (6.8) of D, the rules

$$i\!:\!A_0 \Leftarrow i\!:\!A_1 \wedge \cdots \wedge i\!:\!A_n \quad (i = 0, 1)$$

for the static laws (6.8) of D, the rules

$$1\!:\!A_0 \Leftarrow 0\!:\!A_1 \wedge \cdots \wedge 0\!:\!A_m \wedge 1\!:\!A_{m+1} \wedge \cdots \wedge 1\!:\!A_n$$

for the dynamic laws (6.2) of D, the rules

$$\begin{gathered} 0\!:\!a = \mathbf{t} \Leftarrow 0\!:\!a = \mathbf{t}, \\ 0\!:\!a = \mathbf{f} \Leftarrow 0\!:\!a = \mathbf{f} \end{gathered}$$

for all action constants a, and the rules

$$0\!:\!f = v \Leftarrow 0\!:\!f = v$$

for all regular fluents f and all $v \in \mathit{Dom}(f)$.

We will prove claim (b); the proof of claim (a) is similar. The proof is based on the fact that program $PN_1(D)$ is tight,[1] so that its stable models are characterized by its completion. On the other hand, the models of the causal

[1]See, for instance, Section 2.4.4.

theory $D^1_{\mathcal{C}}$ are characterized by its completion in the sense of [33, Section 2.6]. We will calculate and compare these two completions.

The completion of $D^1_{\mathcal{C}}$ is the conjunction of formulas of four kinds. For every atom A_0 containing a regular fluent, it includes the equivalence

$$0\!:\!A_0 \;\leftrightarrow\; \bigvee\left(\bigwedge_{i=1}^{n} 0\!:\!A_i\right) \;\vee\; 0\!:\!A_0, \tag{7.56}$$

where the big disjunction extends over all static laws (6.1) of D beginning with that atom. Similarly, for every atom A_0 containing a statically determined fluent, it includes the equivalence

$$0\!:\!A_0 \;\leftrightarrow\; \bigvee\left(\bigwedge_{i=1}^{n} 0\!:\!A_i\right). \tag{7.57}$$

For every atom A_0, it includes the equivalence

$$1\!:\!A_0 \;\leftrightarrow\; \bigvee\left(\bigwedge_{i=1}^{n} 1\!:\!A_i\right) \vee \bigvee\left(\bigwedge_{i=1}^{m} 0\!:\!A_i \wedge \bigwedge_{i=m+1}^{n} 1\!:\!A_i\right), \tag{7.58}$$

where the first big disjunction extends over all static laws (6.1) of D beginning with that atom, and the second big disjunction extends over all dynamic laws (6.2) of D beginning with that atom. Finally, for every action constant a it includes the equivalences

$$\begin{array}{c} 0\!:\!a = \mathbf{t} \leftrightarrow 0\!:\!a = \mathbf{t}, \\ 0\!:\!a = \mathbf{f} \leftrightarrow 0\!:\!a = \mathbf{f}. \end{array} \tag{7.59}$$

These equivalences are tautological and can be disregarded.

On the other hand, $PN_1(D)$ is a tight logic program: since $m = 0$ for all static laws (6.1), time stamps decrease along any path in its dependency graph. The program's completion includes formulas similar to (7.56)–(7.59):

$$0\!:\!A_0 \;\leftrightarrow\; \bigvee\left(\bigwedge_{i=1}^{n} 0\!:\!\neg\neg A_i\right) \vee 0\!:\!A_0 \tag{7.60}$$

for each atom A_0 containing a regular fluent,

$$0\!:\!A_0 \;\leftrightarrow\; \bigvee\left(\bigwedge_{i=1}^{n} 0\!:\!\neg\neg A_i\right) \tag{7.61}$$

for each atom A_0 containing a statically determined fluent,

$$1 : A_0 \leftrightarrow \bigvee\left(\bigwedge_{i=1}^{n} \neg\neg\ 1\!:\!A_i\right) \vee \bigvee\left(\bigwedge_{i=1}^{m} 0\!:\!A_i \wedge \bigwedge_{i=m+1}^{n} \neg\neg\ 1\!:\!A_i\right) \tag{7.62}$$

for each atom A_0, and

$$0\!:\!a \leftrightarrow 0\!:\!a \tag{7.63}$$

for each action constant a. In addition, the completion of $PN_1(D)$ includes formulas corresponding to constraints:

$$\neg\left(\bigwedge_{v\in Dom(f)} \neg i\!:\!(f = v)\right) \tag{7.64}$$

for each fluent f and $i = 0, 1$, and

$$\neg(i\!:\!(f = v) \wedge i\!:\!(f = w)) \tag{7.65}$$

for each fluent f, each pair of distinct members v, w of its domain, and $i = 0, 1$. The equivalences (7.63) are tautological and can be disregarded. Take any multi-valued interpretations s_0, s_1 of fluent constants and any set α of action constants such that the triple (7.54) is a transition of $D_{\mathbb{C}}$. Then the multi-valued interpretation (7.55) satisfies the completion (7.56)–(7.58) of $D^1_{\mathbb{C}}$ [33, Proposition 6]. Consequently the interpretation $0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1$ satisfies the formulas (7.60)–(7.62) from the completion of $PN_1(D)$. It obviously satisfies (7.64) and (7.65) as well. It follows that this interpretation is a stable model of $PN_1(D)$, so that $\langle s_0, \alpha, s_1\rangle$ is a transition of D.

The other way around, for any transition $\langle s_0, \alpha, s_1\rangle$ of D, the interpretation $0\!:\!s_0 \cup 0\!:\!\alpha \cup 1\!:\!s_1$ is a stable model of $PN_1(D)$, and consequently satisfies the completion (7.60)–(7.62), (7.64), (7.65) of this program. In view of the last two formulas, (7.55) is a multi-valued interpretation; in view of (7.60)–(7.62), this multi-valued interpretation satisfies the completion (7.56)–(7.58) of $D^1_{\mathbb{C}}$. Consequently it is a model of $D^1_{\mathbb{C}}$, so that (7.54) is a transition of $D_{\mathbb{C}}$.

Chapter 8

Lloyd-Topor Completion and General Stable Models

Section 2.4.4 describes a case when the stable model semantics is equivalent to program completion. Let F be the program (2.23), that is

$$\begin{array}{l} p(a), \\ q(b), \\ p(x) \leftarrow q(x), \end{array}$$

or, in other words, the sentence

$$p(a) \wedge q(b) \wedge \forall x(q(x) \rightarrow p(x)).$$

The program is tight and Fact 2.4.5 shows that its stable models are described as the conjunction of the completed definitions of p and q, which is (2.24):

$$\begin{array}{l} \forall x(p(x) \leftrightarrow x = a \vee q(x)), \\ \forall x(q(x) \leftrightarrow x = b). \end{array}$$

Let now F be the program (2.25), that is,

$$\begin{array}{l} p(x) \leftarrow q(x), \\ q(a) \leftarrow p(b). \end{array}$$

This program is not tight in the sense of Section 2.4.4, so that the above-mentioned theorem is not applicable. In fact, SM$[F]$ is stronger in this case than the conjunction of the completed definitions

$$\begin{array}{l} \forall x(p(x) \leftrightarrow q(x)), \\ \forall x(q(x) \leftrightarrow x = a \wedge p(b)). \end{array} \tag{8.1}$$

A counterexample is provided by any interpretation that treats each of the symbols p, q as a singleton such that its element is equal to both a and b. Such a (non-Herbrand) interpretation satisfies (8.1), but it is not a stable model of (2.25). (In stable models of (2.25) both p and q are empty.)

Program (2.25) is, however, atomic-tight in the sense of [41, Section 5.1.1]. Corollary 5 from that paper allows us to conclude that the equivalence between SM[F] and (8.1) is entailed by the unique name assumption $a \neq b$. It follows that the result of applying SM to the program obtained from (8.1) by adding the constraint

$$\leftarrow a = b$$

is equivalent to the conjunction of the completion sentences (8.1) with $a \neq b$. This example illustrates the role of a property more general than the logical equivalence between SM[F] and the completion of F: it may be useful to know when the equivalence between these two formulas is entailed by a certain set of assumptions. This information may be relevant if we are interested in a logic program obtained from F by adding constraints.

The result of applying SM to the program

$$\begin{array}{rl} p(a) & \leftarrow p(b), \\ q(c) & \leftarrow q(d), \\ & \leftarrow a = b, \\ & \leftarrow c = d \end{array} \tag{8.2}$$

is equivalent to the conjunction of the formulas

$$\begin{array}{c}\forall x(p(x) \leftrightarrow x = a \wedge p(b)),\\ \forall x(q(x) \leftrightarrow x = c \wedge q(d)),\\ a \neq b,\\ c \neq d.\end{array} \tag{8.3}$$

This claim cannot be justified, however, by a reference to Corollary 5 from [41]. The program in this example is atomic-tight, but it does not contain constraints corresponding to some of the unique name axioms, for instance $a \neq c$. We will show how our claim follows from Theorem 8.2.1 of this chapter.

We will discuss also an example illustrating limitations of earlier work that is related to describing dynamic domains in answer set programming. The program in that example is not atomic-tight because of rules expressing the commonsense law of inertia. We will show nevertheless that the process of completion can be used to characterize its stable models by a first-order formula.

The class of tight programs is defined in [20] in terms of predicate dependency graphs; that definition is reproduced in Section 2.4.4 below. The definition of an atomic-tight program in [41] refers to more informative "first-order dependency graphs." Our approach is based on an alternative solution to the problem of making predicate dependency graphs more informative, "rule dependency graphs."

We begin with defining rule dependency graphs in Section 8.1, state the main theorem of this chapter (Theorem 8.2.1) and give examples of its use in Sections 8.2 and 8.3.

8.1 Rule Dependency Graph

We are interested in conditions on a Lloyd-Topor program Π ensuring that the equivalence

$$\mathrm{SM}[\Pi] \leftrightarrow \mathrm{Comp}[\Pi]$$

is entailed by a given set of assumptions Γ. Fact 2.4.5 gives a solution for the special case when Γ is empty. The following definition will help us answer the more general question.

The *rule dependency graph* of a Lloyd-Topor program Π is the directed graph that has

- rules of Π, with variables (both free and bound) renamed arbitrarily, as its vertices, and
- an edge from a rule $p(\mathbf{t}) \leftarrow G$ to a rule $p'(\mathbf{t}') \leftarrow G'$, labeled by an atomic formula $p'(\mathbf{s})$, if $p'(\mathbf{s})$ has a positive nonnegated occurrence in G.

Unlike the predicate dependency graph, the rule dependency graph of a program is usually infinite. For example, the rule dependency graph of program (2.26), that is,

$$\begin{array}{l} p(a,b) \\ q(x,y) \leftarrow p(y,x) \wedge \neg p(x,y) \end{array}$$

has the vertices $p(a,b)$ and

$$q(x_1,y_1) \leftarrow p(y_1,x_1) \wedge \neg p(x_1,y_1) \tag{8.4}$$

for arbitrary pairs of distinct variables x_1, y_1. It has an edge from each vertex (8.4) to $p(a, b)$, labeled $p(y_1, x_1)$. The rule dependency graph of program (2.27) has edges of two kinds:

- from $p(x_1) \leftarrow q(x_1)$ to $q(x_2) \leftarrow r(x_2)$, labeled $q(x_1)$, and
- from $q(x_1) \leftarrow r(x_1)$ to $r(x_2) \leftarrow s(x_2)$, labeled $r(x_1)$

for arbitrary variables x_1, x_2.

The rule dependency graph of a program is "dual" to its predicate dependency graph, in the following sense. The vertices of the predicate dependency graph are predicate symbols, and the presence of an edge from p to q is determined by the existence of a rule that contains certain occurrences of p and q. The vertices of the rule dependency graph are rules, and the presence of an edge from R_1 to R_2 is determined by the existence of a predicate symbol with certain occurrences in R_1 and R_2.

There is a simple characterization of tightness in terms of rule dependency graphs:

Proposition 8.1.1. *A Lloyd-Topor program* Π *is tight iff there exists* n *such that the rule dependency graph of* Π *has no paths of length* n.

Proof. Assume that Π is tight, and let n be the number of predicate symbols occurring in Π. Then the rule dependency graph of Π has no paths of length $n+1$. Indeed, assume that such a path exists:

$$R_0 \xrightarrow{p_1(\dots)} R_1 \xrightarrow{p_2(\dots)} R_2 \xrightarrow{p_3(\dots)} \cdots \xrightarrow{p_{n+1}(\dots)} R_{n+1}.$$

Each of the rules R_i $(1 \leq i \leq n)$ contains p_i in the head and a positive nonnegated occurrence of p_{i+1} in the body. Consequently the predicate dependency graph of Π has an edge from p_i to p_{i+1}, so that $p_1, \dots, p_{n+1}$ is a path in that graph; contradiction. Now assume that Π is not tight. Then there is an infinite path $p_1, p_2, \dots$ in the predicate dependency graph of Π. Let R_i be a rule of Π that has p_i in the head and a positive nonnegated occurrence of p_{i+1} in the body. Then the rule dependency graph of Π has an infinite path of the form

$$R_1 \xrightarrow{p_2(\dots)} R_2 \xrightarrow{p_3(\dots)} \cdots \ .$$

Theorem 8.2.1, stated in the next section, refers to finite paths in the rule dependency graph of a program Π that satisfy an additional condition: the rules at their vertices have no common variables (neither free nor bound). Such paths will be called *chains*.

Corollary 8.1.2. *A Lloyd-Topor program* Π *is tight iff there exists* n *such that* Π *has no chains of length* n*.*

Indeed, any finite path in the rule dependency graph of Π can be converted into a chain of the same length by renaming variables.

8.2 Main Theorem on Γ-Tightness

Let C be a chain

$$\begin{array}{c} p_0(\mathbf{t}^0) \leftarrow Body_0 \\ \downarrow p_1(\mathbf{s}^1) \\ p_1(\mathbf{t}^1) \leftarrow Body_1 \\ \downarrow p_2(\mathbf{s}^2) \\ \cdots\cdots\cdots\cdots \\ \downarrow p_n(\mathbf{s}^n) \\ p_n(\mathbf{t}^n) \leftarrow Body_n \end{array} \tag{8.5}$$

in a Lloyd-Topor program Π. The corresponding *chain formula* F_C is the conjunction

$$\bigwedge_{i=1}^{n} \mathbf{s}^i = \mathbf{t}^i \wedge \bigwedge_{i=0}^{n} Body_i.$$

For instance, if C is the chain

$$\begin{array}{c} q(x_1, y_1) \leftarrow p(y_1, x_1) \wedge \neg p(x_1, y_1) \\ \downarrow p(y_1, x_1) \\ p(a, b) \end{array}$$

in program (2.26) then F_C is

$$y_1 = a \wedge x_1 = b \wedge p(y_1, x_1) \wedge \neg p(x_1, y_1).$$

Let Γ be a set of sentences. About a Lloyd-Topor program Π we will say that it is *tight relative to* Γ, or Γ-*tight*, if there exists a positive integer n such that, for every chain C in Π of length n,

$$\Gamma, \mathrm{Comp}[\Pi] \models \widetilde{\forall} \neg F_C.$$

Theorem 8.2.1. *If a Lloyd-Topor program* Π *is* Γ-*tight then*

$$\Gamma \models \mathrm{SM}[\Pi] \leftrightarrow \mathrm{Comp}[\Pi].$$

The proof of the theorem is given in Chapter 9. The proof is based on the theory of stable models of infinitary propositional formulas [72].

Corollary 8.1.2 shows that every tight program is trivially Γ-tight even when Γ is empty. Consequently Theorem 8.2.1 can be viewed as a generalization of Fact 2.4.5.

Tightness in the sense of Section 2.4.4 is a syntactic condition that is easy to verify; Γ-tightness is not. Nevertheless, the main theorem is useful because it may allow us to reduce the problem of characterizing the stable models of a program by a first-order formula to verifying an entailment in first-order logic.

Here are some examples. In each case, to verify Γ-tightness we take $n = 1$. We will check the entailment in the definition of Γ-tightness by deriving a contradiction from (some subset of) the assumptions Γ, Comp[Π], and F_C.

Example 8.2.1. The one-rule program

$$p(a) \leftarrow p(x) \wedge x \neq a$$

is tight relative to $\emptyset$. Indeed, any chain of length 1 has the form

$$\begin{array}{c} p(a) \leftarrow p(x_1) \wedge x_1 \neq a \\ \big\downarrow p(x_1) \\ p(a) \leftarrow p(x_2) \wedge x_2 \neq a. \end{array}$$

The corresponding chain formula

$$x_1 = a \wedge p(x_1) \wedge x_1 \neq a \wedge p(x_2) \wedge x_2 \neq a.$$

is contradictory.

Thus the stable models of this program are described by its completion, even though the program is not tight (and not even atomic-tight).

Example 8.2.2. Let Π be the program consisting of the first 2 rules of (8.2):

$$\begin{array}{c} p(a) \leftarrow p(b), \\ q(c) \leftarrow q(d). \end{array}$$

To justify the claim about (8.2) made in the introduction, we will check that Π is tight relative to $\{a \neq b, c \neq d\}$. There are two chains of length 1:

$$\begin{tikzcd} p(a) \leftarrow p(b) \arrow[d, "p(b)"] \\ p(a) \leftarrow p(b) \end{tikzcd}$$

and

$$\begin{tikzcd} q(c) \leftarrow q(d) \arrow[d, "q(d)"] \\ q(c) \leftarrow q(d). \end{tikzcd}$$

The corresponding chain formulas are

$$b = a \wedge p(b) \wedge p(b)$$

and

$$d = c \wedge q(d) \wedge q(d).$$

Each of them contradicts Γ.

Example 8.2.3. Let us check that program (2.25) is tight relative to $\{a \neq b\}$. Its chains of length 1 are

$$\begin{tikzcd} p(x_1) \leftarrow q(x_1) \arrow[d, "q(x_1)"] \\ q(a) \leftarrow p(b) \end{tikzcd}$$

and

$$\begin{tikzcd}
q(a) \leftarrow p(b) \arrow[d, "q(b)"] \\
p(x_1) \leftarrow q(x_1)
\end{tikzcd}$$

for an arbitrary variable x_1. The corresponding chain formulas include the conjunctive term $p(b)$. Using the completion (8.1) of the program, we derive $b = a$, which contradicts Γ.

8.3 A Larger Example

Programs found in actual application of ASP usually involve constructs that are now allowed in Lloyd-Topor programs, such as choice rules and constraints. Nevertheless, Theorem 8.2.1 stated above can help us characterize the stable models of a "realistic" program by a first-order formula.

Example 8.3.1 (Example 2.4.3, continued)**.** Program M in Example 2.4.3 is not atomic-tight, so that methods of [41] are not directly applicable to it. Nevertheless, we can describe the stable models of this program without the use of second-order quantifiers. In the statement of the proposition below, **p** stands for the list of intensional predicates *step*, *next* and *at*, and H is the conjunction of the universal closures of the formulas

$$\begin{array}{l}
\widehat{i} \neq \widehat{j} \qquad (1 \leq i < j \leq k),\\
at(x,y,z) \rightarrow object(x) \wedge place(y) \wedge step(z),\\
move(x,y,z) \rightarrow object(x) \wedge place(y) \wedge step(z),\\
at(x,y_1,z) \wedge at(x,y_2,z) \rightarrow y_1 = y_2,\\
object(x) \wedge step(z) \rightarrow \exists y \; at(x,y,z).
\end{array}$$

Proposition 8.3.1. *$SM_{\mathbf{p}}[M]$ is equivalent to the conjunction of H with the* *universal closures of the formulas*

$$step(z) \leftrightarrow \bigvee_{i=0}^{k} z = \widehat{i}, \tag{8.6}$$

$$next(z,u) \leftrightarrow \bigvee_{i=0}^{k-1} (z = \widehat{i} \wedge u = \widehat{i{+}1}), \tag{8.7}$$

$$at(x,y,\widehat{i{+}1}) \leftrightarrow (move(x,y,\widehat{i}) \vee (at(x,y,\widehat{i}) \wedge \neg\exists w\ move(x,w,\widehat{i}))) \quad (i = 0,\dots,k-1). \tag{8.8}$$

Recall that the effect of adding a constraint to a logic program is to eliminate its stable models that violate that constraint [20, Theorem 3]. An interpretation satisfies H iff it does not violate any of the constraints (ii)–(v). So the statement of Proposition 8.3.1 can be summarized as follows: the contribution of rules (i) and (vi)–(viii), under the stable model semantics, amounts to providing explicit definitions for *step* and *next*, and "successor state formulas" for *at*.

The proof of Proposition 8.3.1 refers to the Lloyd-Topor program Π consisting of rules (i), (vi),

(vii′) $at(x,y,0) \leftarrow object(x) \wedge place(y) \wedge \neg\neg at(x,y,0)$,

(viii′) $at(x,y,u) \leftarrow at(x,y,z) \wedge next(z,u) \wedge \neg\neg at(x,y,t_2)$,

and

$$\begin{aligned} object(x) &\leftarrow \neg\neg object(x),\\ place(y) &\leftarrow \neg\neg place(y),\\ move(x,y,z) &\leftarrow \neg\neg move(x,y,z). \end{aligned} \tag{8.9}$$

It is easy to see that $\mathrm{SM}_{\mathbf{p}}[M]$ is equivalent to $\mathrm{SM}[\Pi] \wedge H$. Indeed, consider the program M' obtained from M by adding rules (8.9). These rules are strongly equivalent to the choice rules

$$\{object(x)\},\ \{place(y)\},\ \{move(x, y, z)\}.$$

Consequently $\mathrm{SM}_{\mathbf{p}}[M]$ is equivalent to $\mathrm{SM}[M']$ [20, Theorem 2]. It remains to notice that (vii) is strongly equivalent to (vii′), and (viii) is strongly equivalent to (viii′).

Furthermore—and this is the key step in the proof of Proposition 8.3.1—the second-order formula $\mathrm{SM}[\Pi] \wedge H$ is equivalent to the first-order formula $\mathrm{Comp}[\Pi] \wedge H$, in view of Theorem 8.2.1 and the following fact:

Lemma 8.3.2. *Program* Π *is* H*-tight.*

To derive Proposition 8.3.1 from the lemma, we only need to observe that (8.6) and (8.7) are the completed definitions of *step* and *next* in Π, and that the completed definition of *at* can be transformed into (8.8) under assumptions (8.6), (8.7), and H.

Proof of Lemma 8.3.2. Consider a chain in Π of length $k + 2$:

$$R_0 \xrightarrow{p_1(\dots)} R_1 \xrightarrow{p_2(\dots)} \cdots \xrightarrow{p_{k+1}(\dots)} R_{k+1} \xrightarrow{p_{k+2}(\dots)} R_{k+2}. \tag{8.10}$$

Each R_i is obtained from one of the rules (i), (vi), (vii′), (viii′), (8.9) by renaming variables. Each p_i occurs in the head of R_i and has a positive nonnegated occurrence in R_{i-1}. Since there are no nonnegated predicate symbols in the bodies of rules (i) and (8.9), we conclude that $R_0, \dots, R_{k+1}$ are obtained from other rules of Π, that is, from (vi), (vii′), and (viii′). Since the predicate constant in the head of each of these three rules is *at*, each of $p_1, \dots, p_{k+1}$ is the symbol *at*. Since there are no nonnegated occurrences of *at* in the bodies of (vi) and (vii′), we conclude that $R_0, \dots, R_k$ are obtained by renaming variables in (viii′). This means that chain (8.9) has the form

$$\begin{tikzcd}
at(x_0, y_0, u_0) \leftarrow at(x_0, y_0, z_0) \wedge next(z_0, u_0) \wedge \neg\neg at(x_0, y_0, u_0) \arrow[d, "{at(x_0, y_0, z_0)}"] \\
at(x_1, y_1, u_1) \leftarrow at(x_1, y_1, z_1) \wedge next(z_1, u_1) \wedge \neg\neg at(x_1, y_1, u_1) \arrow[d, "{at(x_1, y_1, z_1)}"] \\
\cdots \arrow[d, "{at(x_{k-1}, y_{k-1}, z_{k-1})}"] \\
at(x_k, y_k, u_k) \leftarrow at(x_k, y_k, z_k) \wedge next(z_k, u_k) \wedge \neg\neg at(x_k, y_k, u_k) \arrow[d, "{at(x_k, y_k, z_k)}"] \\
R_{k+1} \arrow[d, "\cdots"] \\
R_{k+2}.
\end{tikzcd}$$

The corresponding chain formula contains the conjunctive terms

$$z_0 = u_1, z_1 = u_2, \dots, z_{k-1} = u_k$$

and

$$next(z_0, u_0), next(z_1, u_1), \dots, next(z_k, u_k).$$

From these formulas we derive

$$next(u_1, u_0), next(u_2, u_1), \dots, next(u_{k+1}, u_k), \tag{8.11}$$

where u_{k+1} stands for z_k. Using the completed definition of *next*, we conclude:

$$u_i = \widehat{0} \vee \cdots \vee u_i = \widehat{k} \qquad (0 \le i \le k+1).$$

Consider the case when

$$u_i = \widehat{j_i} \qquad (0 \leq i \leq k+1)$$

for some numbers $j_0, \dots, j_{k+1} \in \{0, \dots, k\}$. There exists at least one subscript i such that $j_i \neq j_{i+1} + 1$, because otherwise we would have

$$j_0 = j_1 + 1 = j_2 + 2 = \cdots = j_{k+1} + k + 1,$$

which is impossible because $j_0, j_{k+1} \in \{0, \dots, k\}$. By the choice of i, from the completed definition of *next* and the unique name assumption (included in H) we can derive $\neg next(\widehat{j_{i+1}}, \widehat{j_i})$. Consequently $\neg next(u_{i+1}, u_i)$, which contradicts (8.11).

To sum up, in this chapter we proposed a new method for representing SM[F] in the language of first-order logic. It is more general than the approach of [20]. Its relationship with the ideas of [41] requires further study. This method allows us, in particular, to prove the equivalence of some ASP descriptions of dynamic domains to axiomatizations based on successor state axioms.

Chapter 9

Propositional Infinitary Logic Programs

In this chapter we review the definition of a stable model for infinitary formulas [72] and study their relationship to supported models. This will help us prove Theorem 8.2.1.

9.1 Review of Stable Models of Infinitary Formulas

Let $\mathcal{A}$ be a set of propositional atoms. The sets $\mathcal{F}_0, \mathcal{F}_1, \dots$ are defined as follows:

- $\mathcal{F}_0 = \mathcal{A} \cup \{\bot\}$;
- $\mathcal{F}_{i+1}$ consists of expressions $\mathcal{H}^\wedge$ and $\mathcal{H}^\vee$, for all subsets $\mathcal{H}$ of $\mathcal{F}_0 \cup \ldots \cup \mathcal{F}_i$, and of expressions $F \to G$, where $F, G \in \mathcal{F}_0 \cup \ldots \cup \mathcal{F}_i$.

An *infinitary formula* (over $\mathcal{A}$) is an element of $\bigcup_{i=0}^{\infty} \mathcal{F}_i$.

A *(propositional) interpretation* is a subset of $\mathcal{A}$. The satisfaction relation between an interpretation and an infinitary formula is defined in a natural way. The definition of the reduct F^I of a formula F relative to an interpretation I proposed in [16] is extended to infinitary formulas as follows:

- $\bot^I = \bot$.

- For $A \in \mathcal{A}$, $A^I = \bot$ if $I \not\models A$; otherwise $A^I = A$.

- $(\mathcal{H}^\wedge)^I = \bot$ if $I \not\models \mathcal{H}^\wedge$; otherwise $(\mathcal{H}^\wedge)^I = \{G^I : G \in \mathcal{H}\}^\wedge$.

- $(\mathcal{H}^\vee)^I = \bot$ if $I \not\models \mathcal{H}^\vee$; otherwise $(\mathcal{H}^\vee)^I = \{G^I : G \in \mathcal{H}\}^\vee$.

- $(G \to H)^I = \bot$ if $I \not\models G \to H$; otherwise $(G \to H)^I = G^I \to H^I$.

(Note that according to this definition F^I is $\bot$ whenever $I \not\models F$.) An interpretation I is a *stable model* of an infinitary formula F if I is a minimal model of F^I. An interpretation I satisfies F^I iff it satisfies F [72, Proposition 1], so that stable models of F are models of F.

Infinitary formulas are used to encode first-order sentences as follows. For any interpretation I in the sense of first-order logic, let $\mathcal{A}$ be the set of ground atoms formed from the predicate constants of the underlying signature and the "names" ξ^* of elements ξ of the universe $|I|$ of I—new objects constants that are in a 1–1 correspondence with elements of $|I|$. By I^r we denote the set of atoms from $\mathcal{A}$ that are satisfied by I. In the definition below, t^I stands for the value assigned to the ground term t by the interpretation I. The *grounding* of a first-order sentence F relative to I (symbolically, $gr_I(F)$) is the infinitary formula over A constructed as follows:

- $gr_I(\bot) = \bot$.

- $gr_I(p(t_1, \ldots, t_k)) = p((t_1^I)^*, \ldots, (t_k^I)^*)$.

- $gr_I(t_1 = t_2) = \top$, if $t_1^I = t_2^I$, and $\bot$ otherwise.
- If $F = G \wedge H$, $gr_I(F) = gr_I(G) \wedge gr_I(H)$.
- If $F = G \vee H$, $gr_I(F) = gr_I(G) \vee gr_I(H)$.
- If $F = G \rightarrow H$, $gr_I(F) = gr_I(G) \rightarrow gr_I(H)$.
- If $F = \exists x G(x)$, $gr_I(F) = \{gr_I(G(u^*)) \,:\, u \in |I|\}^\vee$.
- If $F = \forall x G(x)$, $gr_I(F) = \{gr_I(G(u^*)) \,:\, u \in |I|\}^\wedge$.

It is easy to check that gr_I is a faithful translation in the following sense: I satisfies a first-order sentence F iff I^r satisfies $gr_I(F)$.

This transformation is also faithful in the sense of the stable model semantics: I satisfies SM$[F]$ iff I^r is a stable model of $gr_I(F)$ [72, Theorem 5]. This is why infinitary formulas can be used for proving properties of the operator SM.

9.2 Fages' Theorem for Infinitary Programs

An *infinitary rule* is an implication $G \rightarrow A$ with with $A \in \mathcal{A}$. We will write it as $A \leftarrow G$ and call A the *head* and G the *body* of the rule. An *infinitary program* is a conjunction of (possibly infinitely many) implications. For instance, if Π is a Lloyd-Topor program then, for any interpretation I, $gr_I(\Pi)$ is an infinitary program. We say that an interpretation I is *supported* by an infinitary program Π if each atom $A \in I$ is the head of a rule $A \leftarrow G$ of Π such that $I \models G$.

The set of *positive nonnegated atoms* of an infinitary formula, denoted by $\text{Pnn}(F)$, and the set of *negative nonnegated atoms* of F, denoted by $\text{Nnn}(F)$, are defined recursively, as follows:

- $\text{Pnn}(\bot) = \emptyset$.
- For $A \in \mathcal{A}$, $\text{Pnn}(A) = \{A\}$.
- $\text{Pnn}(\mathcal{H}^\wedge) = \text{Pnn}(\mathcal{H}^\vee) = \bigcup_{H \in \mathcal{H}} \text{Pnn}(H)$.
- $\text{Pnn}(G \to H) = \begin{cases} \emptyset & \text{if } H = \bot, \\ \text{Nnn}(G) \cup \text{Pnn}(H) & \text{otherwise.} \end{cases}$
- $\text{Nnn}(\bot) = \emptyset$,
- For $A \in \mathcal{A}$, $\text{Nnn}(A) = \emptyset$.
- $\text{Nnn}(\mathcal{H}^\wedge) = \text{Nnn}(\mathcal{H}^\vee) = \bigcup_{H \in \mathcal{H}} \text{Nnn}(H)$.
- $\text{Nnn}(G \to H) = \begin{cases} \emptyset & \text{if } H = \bot, \\ \text{Pnn}(G) \cup \text{Nnn}(H) & \text{otherwise.} \end{cases}$

Let Π be an infinitary program, and I a propositional interpretation. About atoms $A, A' \in I$ we say that A' is a *parent of A relative to* Π *and I* if Π has a rule $A \leftarrow G$ with the head A such that $I \models G$ and A' is a positive nonnegated atom of G. We say that Π is *tight on I* if there is no infinite sequence $A_0, A_1, \ldots$ of elements of I such that for every i, A_{i+1} is a parent of A_i relative to F and I.

The following theorem expresses a Fages-style property of infinitary formulas similar to Theorem 1 from [13].

Theorem (Fages' Theorem for Infinitary Programs). *For any model I of an infinitary program Π such that Π is tight on I, I is stable iff I is supported by Π.*

9.3 Proof of Fages' Theorem for Infinitary Programs

In this section, Π is an arbitrary infinitary program. It is clear that for any model I of Π, the reduct Π^I is the conjunction of (i) the rules $A \leftarrow G^I$ for all rules $A \leftarrow G$ of Π such that $A \in I$, and (ii) tautologies $\bot \leftarrow \bot$. We will disregard these tautologies and think of Π^I as a program.

Lemma 9.3.1. *A model I of Π is supported by Π iff it is supported by Π^I.*

Proof. A model I of Π is supported by Π^I iff for every atom $A \in I$ there exists a rule $A \leftarrow G$ in Π such that $I \models G^I$. By Proposition 1 from [72], $I \models G^I$ iff $I \models G$.

Lemma 9.3.2. *Any stable model of Π is supported by Π.*

Proof. By Lemma 9.3.1, it is sufficient to check that any stable model I of Π is supported by Π^I. Take an atom $A \in I$. Since I is a stable model of Π, I is minimal among the models of Π^I. Therefore $I \setminus \{A\}$ does not satisfy Π^I, that is to say, for some rule $A' \leftarrow G$ of Π such that

$$A' \in I, \tag{9.1}$$

$I \setminus \{A\}$ does not satisfy the corresponding rule $A' \leftarrow G^I$ of Π^I. Then

$$I \setminus \{A\} \models G^I \tag{9.2}$$

and

$$A' \notin I \setminus \{A\}. \tag{9.3}$$

From (9.2), $I \models G$ (because otherwise G^I would be $\bot$), and consequently $I \models G^I$. From (9.1) and (9.3), $A' = A$. Thus $A' \leftarrow G^I$ is a rule of Π^I such that its head is A and its body is satisfied by I.

Lemma 9.3.3. *For any infinitary formula F and any interpretation I,*

$$\text{Pnn}(F^I) \subseteq \text{Pnn}(F), \quad \text{Nnn}(F^I) \subseteq \text{Nnn}(F).$$

Proof: Straightforward, by strong induction on the construction of F (defined as the value of i for which $F \in \mathcal{F}_i$).

Lemma 9.3.4. *For any model I of Π, if Π is tight on I then so is Π^I.*

Proof. Assume that Π^I is not tight on I, and let $A_0, A_1, \dots$ be an infinite sequence of elements of I such that A_{i+1} is a parent of A_i relative to Π^I and I. Consider the rule of Π^I justifying this property. That rule has the form $A \leftarrow G^I$ for some rule $A \leftarrow G$ of Π such that $A \in I$, and it satisfies the following conditions:

$$A = A_i, \quad I \models G^I, \quad A_{i+1} \in I \cap \text{Pnn}(G^I).$$

Then $I \models G$ and, in view of Lemma C,

$$A_{i+1} \in I \cap \mathrm{Pnn}(G^I) \subseteq I \cap \mathrm{Pnn}(G).$$

Consequently, for every i, A_{i+1} is a parent of A_i relative to I and Π, contrary to the assumption that Π is tight on I.

The statement of the following lemma refers to the set of *strictly positive atoms* of an infinitary formula F, denoted by $\mathrm{SPos}(F)$, which is defined as follows:

- $\mathrm{SPos}(\bot) = \emptyset$.
- For $A \in \mathcal{A}$, $\mathrm{SPos}(A) = \{A\}$.
- $\mathrm{SPos}(\mathcal{H}^\wedge) = \bigcup_{H \in \mathcal{H}} \mathrm{SPos}(H)$.
- $\mathrm{SPos}(\mathcal{H}^\vee) = \bigcup_{H \in \mathcal{H}} \mathrm{SPos}(H)$.
- $\mathrm{SPos}(G \to H) = \mathrm{SPos}(H)$.

Lemma 9.3.5. *For any infinitary formula F, $\mathrm{SPos}(F) \subseteq \mathrm{Pnn}(F)$.*

Proof: Straightforward, by induction on the construction of F.

Lemma 9.3.6. *Let I be a model of an infinitary formula F. If F can be represented in the form G^I for some infinitary formula G then any interpretation J such that*

$$I \cap \mathrm{SPos}(F) \subseteq J$$

is a model of F as well.

Proof. By induction on the construction of G we can show that if $I \models G^I$ (or, equivalently, $I \models G$) and $I \cap \mathrm{SPos}(G^I) \subseteq J$ then $J \models G^I$. Consider the more difficult case when G has the form $H_1 \to H_2$. Since $I \models G$, and G^I is $H_1^I \to H_2^I$. We can distinguish between two subcases: (i) $I \not\models H_1$ and (ii) $I \models H_2$. In the first case, H_1^I is $\bot$, so that G^I is tautological, and the assertion $J \models G^I$ is trivial. Assume now that $I \models H_2$. Since

$$I \cap \mathrm{SPos}(G^I) = I \cap \mathrm{SPos}(H_2^I) \subseteq J,$$

we can conclude from the induction hypothesis that $J \models H_2^I$. Consequently $J \models G^I$.

Proof of Fages' Theorem for Infinitary Programs. The only if part is immediate from Lemma 9.3.2. Let I be a supported model of Π such that Π is tight on I. To prove the stability of I, we need to show that no proper subset of I satisfies Π^I. Take a proper subset J of I. There is an atom A in $I \setminus J$ that has no parent in $I \setminus J$ relative to Π^I and I. Indeed, if every atom in $I \setminus J$ has a parent relative to Π^I and I that belongs to $I \setminus J$ then there exists an infinite sequence $A_0, A_1, \dots$ of elements of $I \setminus J$ such that A_{i+1} is a parent of A_i, so that Π^I is not tight on I; this is impossible by Lemma 9.3.4. Consider such an atom A. By Lemma 9.3.1, I is supported by Π^I. It follows that there is a rule $A \leftarrow F$ in Π^I such that $I \models F$. By the definition of the parent relation, all elements of $I \cap \mathrm{Pnn}(F)$ are parents of A relative to Π^I and I. By the choice of A, no parent of A relative to Π^I and I belongs to $I \setminus J$.

Consequently $I \cap \mathrm{Pnn}(F)$ is disjoint from $I \setminus J$, so that

$$I \cap \mathrm{Pnn}(F) \subseteq J.$$

In view of Lemma 9.3.5, it follows that

$$I \cap \mathrm{SPos}(F) \subseteq J.$$

Since $A \leftarrow F$ is a rule of Π^I, F has the form G^I for some formula G. By Lemma 9.3.6, it follows that $J \models F$. Since $A \in I \setminus J$, we conclude that J does not satisfy $A \leftarrow F$ and therefore is not a model of Π^I.

9.4 Proof of Theorem 8.2.1

In the statement of Theorem 8.2.1, the implication left-to-right

$$\mathrm{SM}[\Pi] \rightarrow \mathrm{Comp}[\Pi]$$

is logically valid for any Lloyd-Topor program Π. This fact follows from [20, Theorem 11] by the argument used in the proof of Fact 2.4.5. To prove the theorem in the other direction, we need to establish the following:

$$\begin{array}{c}\textit{If a Lloyd-Topor program } \Pi \textit{ is } \Gamma\textit{-tight,}\\ \textit{and an interpretation } I \textit{ satisfies both } \Gamma \textit{ and Comp}[\Pi],\\ \textit{then } I \textit{ satisfies SM}[\Pi].\end{array} \tag{9.4}$$

This assertion follows from Fages' theorem for infinitary programs (Section 9.3) and two lemmas. One of them (Lemma 9.4.2) relates the Γ-tightness condition from the statement of Theorem 8.2.1 to tightness on an interpretation as defined in Section 9.2. The other (Lemma 9.4.3) states that models of Comp[Π] can be characterized in terms of satisfaction and supportedness.

9.4.1 Relating Γ-tightness to Tightness on an Interpretation

Our goal is to relate the Γ-tightness of a Lloyd-Topor program Π (defined in Section 8.2) to the tightness of $gr_I(\Pi)$ on I^r in the sense of Section 9.4. As a preliminary step, we will describe a relationship between positive nonnegated atomic subformulas of a first-order formula F, referred to in the definition of the rule dependency graph, and the positive nonnegated atoms of the infinitary formula $gr_I(F)$.

In the following lemmas, I is an interpretation in the sense of first-order logic, and F is a first-order sentence that may contain the names ξ^* of elements ξ of the universe of I. If $\mathbf{u}$ is a tuple $\xi_1, \ldots, \xi_k$ of elements of the universe then $\mathbf{u}^*$ stands for the corresponding tuple of names $\xi_1^*, \ldots, \xi_k^*$. If $\mathbf{t}$ is a tuple $t_1, \ldots, t_k$ of ground terms then $gr_I(\mathbf{t})$ stands for the tuple $(t_1^I)^*, \ldots, (t_k^I)^*$ of the names of their values.

Lemma 9.4.1. *For any ground atom of the form $p(\boldsymbol{u}^*)$,*

(i) if $p(\boldsymbol{u}^*) \in \text{Pnn}(gr_I(F))$ then $\boldsymbol{u}^*$ has the form $gr_I(\boldsymbol{t}(\mathbf{v}^*))$ for some tuple $\boldsymbol{t}(\boldsymbol{x})$ of terms such that $p(\boldsymbol{t}(\boldsymbol{x}))$ has a positive nonnegated occurrence in F, and some tuple $\mathbf{v}$ of elements of the universe;

(ii) if $p(\boldsymbol{u}^*) \in \text{Nnn}(gr_I(F))$ then $\boldsymbol{u}^*$ has the form $gr_I(\boldsymbol{t}(\mathbf{v}^*))$ for some tuple $\boldsymbol{t}(\boldsymbol{x})$ of terms such that $p(\boldsymbol{t}(\boldsymbol{x}))$ has a negative nonnegated occurrence in F, and some tuple $\mathbf{v}$ of elements of the universe.

Proof. The proof is by induction on the size of F. We will consider three cases: when F atomic, when F is an implication, and when F begins with the universal quantifier.

If F is an atomic formula that does not contain p then $gr(F)$ does not contain atoms of the form $p(\mathbf{u}^*)$, and assertions (i) and (ii) are trivial. Assume that F is $p(\mathbf{t})$, so that $gr_I(p(\mathbf{t})) = p(gr_I(\mathbf{t}))$, $\mathrm{Pnn}(gr_I(F)) = \{p(gr_I(\mathbf{t}))\}$, and $\mathrm{Nnn}(gr_I(F)) = \emptyset$. If $p(\mathbf{u}^*) \in \mathrm{Pnn}(gr_I(F))$ then $\mathbf{u}^* = gr_I(\mathbf{t})$; $p(\mathbf{u}^*) \in \mathrm{Nnn}(gr_I(F))$ is impossible.

If F is $G \to H$ then $gr_I(F)$ is $gr_I(G) \to gr_I(H)$. Assume that $gr_I(H)$ is different from $\bot$ (otherwise both $\mathrm{Pnn}(gr_I(F))$ and $\mathrm{Nnn}(gr_I(F))$ are empty). Then

$$\begin{aligned}\mathrm{Pnn}(gr_I(F)) &= \mathrm{Nnn}(gr_I(G)) \cup \mathrm{Pnn}(gr_I(H)),\\ \mathrm{Nnn}(gr_I(F)) &= \mathrm{Pnn}(gr_I(G)) \cup \mathrm{Nnn}(gr_I(H)).\end{aligned}$$

To prove (i), assume that $p(\mathbf{u}^*) \in \mathrm{Pnn}(gr_I(F))$. Then

$$p(\mathbf{u}^*) \in \mathrm{Nnn}(gr_I(G)) \ \text{ or } \ p(\mathbf{u}^*) \in \mathrm{Pnn}(gr_I(H)).$$

By the induction hypothesis, it follows that $\mathbf{u}^*$ has the form $gr_I(\mathbf{t}(\mathbf{v}^*))$ for some tuple $\mathbf{t}(\mathbf{x})$ of terms such that $p(\mathbf{t}(\mathbf{x}))$ has a negative nonnegated occurrence in G or a positive nonnegated occurrence in H. Since $gr_I(H)$ is not $\bot$, H is not $\bot$ either. Consequently $p(\mathbf{t}(\mathbf{x}))$ has a positive nonnegated occurrence in $G \to H$. The proof of (ii) is similar.

If F is $\forall z G(z)$ then

$$gr_I(F) = \{gr_I(G(w^*)) \,:\, w \in |I|\}^{\wedge}.$$

To prove (i), assume that $p(\mathbf{u}^*) \in \text{Pnn}(gr_I(F))$. Since

$$\text{Pnn}(gr_I(F)) = \bigcup_{w \in |I|} \text{Pnn}(gr_I(G(w^*))),$$

$p(\mathbf{u}^*) \in \text{Pnn}(gr_I(G(w^*)))$ for some $w \in |I|$. By the induction hypothesis, it follows that $\mathbf{u}^*$ has the form $gr_I(\mathbf{t}(\mathbf{v}^*))$ for some tuple $\mathbf{t}(\mathbf{x})$ of terms such that, for some $w \in |I|$, $p(\mathbf{t}(\mathbf{x}))$ has a positive nonnegated occurrence in $G(w^*)$. Without loss of generality we can assume that every member of $\mathbf{x}$ occurs in $\mathbf{t}(\mathbf{x})$. *Case 1:* z is not a member of $\mathbf{x}$. Let $p(\mathbf{t}'(\mathbf{x}, z))$ be the part of $G(z)$ from which the occurrence of $p(\mathbf{t}(\mathbf{x}))$ in $G(w^*)$ is obtained by substituting w^* for z. This part has a positive nonnegated occurrence in $G(z)$, and consequently in $F(z)$. On the other hand, $t(\mathbf{x})$ is $t'(\mathbf{x}, w^*)$, so that $t(\mathbf{v}^*)$ is $t'(\mathbf{v}^*, w^*)$, and

$$\mathbf{u}^* = gr_I(t(\mathbf{v}^*)) = gr_I(\mathbf{t}'(\mathbf{v}^*, w^*)).$$

Case 2: z is a member of $\mathbf{x}$. Then $p(\mathbf{t}(\mathbf{x}))$ contains z, which is only possible if all occurrences of z in the part of $F(z)$ from which the occurrence of $p(\mathbf{t}(\mathbf{x}))$ is obtained by substitution are bound. Then that part of $F(z)$ is not affected by the substitution and equals $p(\mathbf{t}(\mathbf{x}))$. Thus $p(\mathbf{t}(\mathbf{x}))$ has a positive nonnegated occurrence in $F(z)$, and $\mathbf{u}^*$ is $gr_I(t(\mathbf{v}^*))$. The proof of part (ii) is similar.

Lemma 9.4.2. *If a Lloyd-Topor program Π is Γ-tight, and an interpretation I satisfies both Γ and Comp$[\Pi]$, then $gr_I(\Pi)$ is tight on I^r.*

Proof. Assume that Π is Γ-tight, that an interpretation I satisfies both Comp$[\Pi]$ and Γ, and that $gr_I(\Pi)$ is not tight on I^r. Then there exists an infinite sequence $A_0, A_1, \dots$ of atoms such that each A_{i+1} is a parent of A_i relative to $gr_I(\Pi)$ and I^r. In other words, there exist rules

$$p_i(gr_I(\mathbf{t}^i(\mathbf{c}_i^*))) \leftarrow gr_I(G_i(\mathbf{c}_i^*)) \qquad (i = 0, 1, \dots)$$

of $gr_I(\Pi)$, obtained by grounding from rules

$$p_i(\mathbf{t}^i(\mathbf{x}^i)) \leftarrow G_i(\mathbf{x}^i) \tag{9.5}$$

of Π, such that A_i is $p_i(gr_I(\mathbf{t}^i(\mathbf{c}_i^*)))$,

$$I^r \models gr_I(G_i(\mathbf{c}_i^*)), \tag{9.6}$$

and $A_{i+1} \in \text{Pnn}(gr_I(G_i(\mathbf{c}_i^*)))$. Atom A_{i+1} can be written as $p_{i+1}(\mathbf{u}^*)$, where $\mathbf{u}^*$ is $gr_I(\mathbf{t}^{i+1}(\mathbf{c}_{i+1}^*))$. By Lemma 9.4.1,

$$gr_I(\mathbf{t}^{i+1}(\mathbf{c}_{i+1}^*)) \text{ is } gr_I(\mathbf{s}^{i+1}(\mathbf{d}_i^*)) \tag{9.7}$$

for some atom $p_{i+1}(\mathbf{s}^{i+1}(\mathbf{z}^i))$ that has a positive nonnegated occurrence in $G_i(\mathbf{c}_i^*)$, and some tuple $\mathbf{d}_i$ of elements of the universe. That occurrence of $p_{i+1}(\mathbf{s}^{i+1}(\mathbf{z}^i))$ is the result of substituting $\mathbf{c}_i^*$ for $\mathbf{x}^i$ in some atom $p_{i+1}(\mathbf{r}^{i+1}(\mathbf{x}^i, \mathbf{z}^i))$ that has a positive nonnegated occurrence in $G_i(\mathbf{x}^i)$, so that $\mathbf{s}^{i+1}(\mathbf{z}^i)$ is $\mathbf{r}^{i+1}(\mathbf{c}_i^*, \mathbf{z}^i)$. From (9.7) we conclude that

$$gr_I(\mathbf{t}^{i+1}(\mathbf{c}_{i+1}^*)) \text{ is } gr_I(\mathbf{r}^{i+1}(\mathbf{c}_i^*, \mathbf{d}_i^*)). \tag{9.8}$$

Since Π is Γ-tight and I satisfies Comp[Π] and Γ, there exists n such that, for every chain C in Π of length n, $I \models \widetilde{\forall} \neg F_C$. Consider rules (9.5) for $i = 0, \dots, n$. Let

$$p_i(\mathbf{t}^i(\widehat{\mathbf{x}^i})) \leftarrow \widehat{G_i}(\widehat{\mathbf{x}^i}) \tag{9.9}$$

be those rules with variables renamed so that different rules have no common variables. (Formula $\widehat{G_i}(\widehat{\mathbf{x}^i})$ is the result of renaming bound variables in $G_i(\widehat{\mathbf{x}^i})$.) Then $p_{i+1}(\mathbf{r}^{i+1}(\widehat{\mathbf{x}^i}, \widehat{\mathbf{z}^i}))$ has a positive nonnegated occurrence in $\widehat{G_i}(\widehat{\mathbf{x}^i})$, for some tuple $\widehat{\mathbf{z}^i}$ of variables. Let C be the chain

$$p^0(\mathbf{t}^0(\widehat{\mathbf{x}^0})) \leftarrow \widehat{G^0}(\widehat{\mathbf{x}^0})$$

$$\downarrow p^1(\mathbf{r}^1(\widehat{\mathbf{x}^0}, \widehat{\mathbf{z}^0}))$$

$$p^1(\mathbf{t}^1(\widehat{\mathbf{x}^1})) \leftarrow \widehat{G^1}(\widehat{\mathbf{x}^1})$$

$$\downarrow p^2(\mathbf{r}^2(\widehat{\mathbf{x}^1}, \widehat{\mathbf{z}^1}))$$

$$p^2(\mathbf{t}^2(\widehat{\mathbf{x}^2})) \leftarrow \widehat{G^2}(\widehat{\mathbf{x}^2})$$

$$\downarrow p^3(\mathbf{r}^3(\widehat{\mathbf{x}^2}, \widehat{\mathbf{z}^2}))$$

$$\cdots$$

$$\downarrow p^n(\mathbf{r}^n(\widehat{\mathbf{x}^{n-1}}, \widehat{\mathbf{z}^{n-1}}))$$

$$p^n(\mathbf{t}^n(\widehat{\mathbf{x}^n})) \leftarrow \widehat{G^n}(\widehat{\mathbf{x}^n}).$$

The corresponding chain formula F_C is

$$\bigwedge_{i=0}^{n-1} \mathbf{t}^{i+1}(\widehat{\mathbf{x}^{i+1}}) = \mathbf{r}^{i+1}(\widehat{\mathbf{x}^i}, \widehat{\mathbf{z}^i}) \wedge \bigwedge_{i=0}^{n} \widehat{G_i}(\widehat{\mathbf{x}^i}).$$

Since interpretation I satisfies $\widetilde{\forall}\neg F_C$, it satisfies also

$$\neg\left(\bigwedge_{i=0}^{n-1} \mathbf{t}^{i+1}(\widehat{\mathbf{c}^*_{i+1}}) = \mathbf{r}^{i+1}(\widehat{\mathbf{c}^*_i}, \widehat{\mathbf{d}^*_i}) \wedge \bigwedge_{i=0}^{n} \widehat{G_i}(\widehat{\mathbf{c}^*_i})\right),$$

so that I^r satisfies

$$\neg\left(\bigwedge_{i=0}^{n-1} gr_I(\mathbf{t}^{i+1}(\widehat{\mathbf{c}^*_{i+1}}) = \mathbf{r}^{i+1}(\widehat{\mathbf{c}^*_i}, \widehat{\mathbf{d}^*_i})) \wedge \bigwedge_{i=0}^{n} gr_I(\widehat{G_i}(\widehat{\mathbf{c}^*_i}))\right).$$

In view of (9.8), each of the formulas $gr_I(\mathbf{t}^{i+1}(\widehat{\mathbf{c}^*_{i+1}}) = \mathbf{r}^{i+1}(\widehat{\mathbf{c}^*_i}, \widehat{\mathbf{d}^*_i}))$ is $\top$, so that I^r satisfies

$$\neg \bigwedge_{i=0}^{n} gr_I(\widehat{G_i}(\widehat{\mathbf{c}^*_i})).$$

This is impossible by (9.6).

9.4.2 Relating Models of Completion to Supportedness

Lemma 9.4.3. *For any Lloyd-Topor program* Π*, an interpretation* I *satisfies* $Comp[\Pi]$ *iff* I^r *satisfies* $gr_I(\Pi)$ *and is supported by* $gr_I(\Pi)$*.*

Proof. Recall that the rules of a Lloyd-Topor program Π have the form

$$p(\mathbf{t}(\mathbf{y})) \leftarrow G(\mathbf{y})$$

(with all free variables of the rule explicitly shown), and that the rules of the infinitary program $gr_I(\Pi)$ have the form

$$p(gr_I(\mathbf{t}(\mathbf{v}^*))) \leftarrow gr_I(G(\mathbf{v}^*)) \tag{9.10}$$

for all tuples $\mathbf{v}$ of elements of $|I|$. For any Lloyd-Topor program Π, Comp[Π] is equivalent to the conjunction of Π with the universal closures of the definitions (2.21) of all predicate constants p. To prove Lemma 9.4.3, we need to check that the condition: for all p,

$$I \models \forall \mathbf{x} \left(p(\mathbf{x}) \rightarrow \bigvee_i \exists \mathbf{y}^i (\mathbf{x} = \mathbf{t}^i(\mathbf{y}^i)) \wedge G^i(\mathbf{y}^i) \right), \tag{9.11}$$

is equivalent to the assertion that $gr_I(\Pi)$ is supported by I^r. Note first that (9.11) is equivalent to the condition:

$$I^r \models \left\{ gr_I \left(p(\mathbf{u}^*) \to \bigvee_i \exists \mathbf{y}^i (\mathbf{u}^* = \mathbf{t}^i(\mathbf{y}^i) \wedge G_i(\mathbf{y}^i)) \right) \,:\, \mathbf{u} \in |I|^k \right\}^{\wedge},$$

where k is the arity of p. The conjunctive terms $gr_I(\cdots)$ can be written as

$$p(\mathbf{u}^*) \to \bigvee_i \left\{ gr_I(\mathbf{u}^* = \mathbf{t}^i(\mathbf{v}^*)) \wedge gr_I(G_i(\mathbf{v}^*))) \,:\, \mathbf{v} \in |I|^{l_i} \right\}^{\vee},$$

where l_i is the length of the tuple $\mathbf{y}^i$. Therefore (9.11) is equivalent to following condition: for every $\mathbf{u} \in |I|^k$ such that $p(\mathbf{u}^*) \in I^r$,

$$\text{there exist } i \text{ and } \mathbf{v} \text{ such that } \mathbf{u}^* \text{ is } gr_I(\mathbf{t}^i(\mathbf{v}^*)), \text{ and } I^r \models gr_I(G_i(\mathbf{v}^*)). \quad (9.12)$$

Condition (9.12) is equivalent to saying that $p(\mathbf{u}^*)$ is the head of one of the rules (9.10) whose body is satisfied by I^r.

9.4.3 Proof of Theorem 8.2.1

To derive assertion (9.4) from these lemmas, assume that Π is a Γ-tight Lloyd-Topor program, and that I is an interpretation satisfying both Γ and Comp[Π]. By Lemma 9.4.2, $gr_I(\Pi)$ is tight on I^r. By Lemma 9.4.3, I^r satisfies $gr_I(\Pi)$ and is supported by $gr_I(\Pi)$. By Fages' theorem for infinitary formulas (Section 9.2), it follows that I^r is a stable model of $gr_I(\Pi)$. By Theorem 5 from [72], quoted at the end of Section 9.1, it follows that I satisfies SM[Π].

Chapter 10

From $\mathcal{BC}$ to the Input Language of CPLUS2ASP

Software system CPLUS2ASP, designed and implemented at Arizona State University, was originally created to answer queries about action descriptions described in $\mathcal{C}+$. According to [2], its version 2 is applicable also to other action languages, including the language $\mathcal{BC}$ described in Chapter 6 of this dissertation. However, there are significant syntactic differences between $\mathcal{BC}$ and the input language of CPLUS2ASP. For instance, in the input language of CPLUS2ASP we can declare sorts, variables and constants; one line in CPLUS2ASP code may correspond to a large set of causal laws of $\mathcal{BC}$ that follow the same pattern.

In this chapter, we clarify the relationship between $\mathcal{BC}$ and a fragment of the input language CPLUS2ASP by describing a translation from the latter to the former. A context-free grammar describing the syntax of that fragment is given in Appendix 1. It is based on the grammar used for the generation of the parser of CPLUS2ASP[1]. In Section 10.1, we give an informal description of that syntax and examples of its use. The translation into $\mathcal{BC}$ is given in

[1]Joseph Babb, personal communication, Jan, 13, 2014

Section 10.2.

10.1 Syntax

10.1.1 Identifiers and Integers

An identifier is a string of letters, digits and underscores that begins with a letter. An integer is a whole number in the usual decimal notation, such as 5 or -20. Extended integers are arithmetic expressions that are evaluated to integers. A number range is defined as an interval of extended integers, such as 1..5.

10.1.2 Comments

A comment is the text appearing on a line following a % or appearing on one or more lines between /* and */.

10.1.3 Declarations

A declaration assigns an identifier, or each member of a group of identifiers, to one of 4 categories: *sorts*, *objects*, *constants*, *variables*. An identifier for a variable must be capitalized. Identifiers for the other three categories should not be capitalized. Every declaration begins with :- and the name of the group.

10.1.3.1 Sort Declarations

A sort represents a set of objects. A sort declaration allows us to describe the subsort relation between sorts. For instance, the declaration

```
:- sorts
  index;
  location >> block.
```

expresses that there are three sets of objects—indices, locations, blocks—and also that every `block` is a `location`.

10.1.3.2 Object Declarations

An object declaration describes expressions of certain syntactic forms as objects and specifies the sorts of these objects. For instance, the declaration

```
:- objects
  table      :: location;
  1..10      :: index;
  box(index) :: block.
```

expresses that `table` is an object of sort `location`, that numbers `1,..,10` are objects of sort `index`, and that the identifier `box` followed by an object of sort `index` enclosed in parentheses is an object of sort `block`.

10.1.3.3 Variable Declarations

A variable declaration describes an identifier, or each of several identifiers, as a schematic variable for objects of a certain sort:

```
:- variables
    P1, P2 :: boolean;
    B1, B2 :: block;
    L1, L2 :: location.
```

(The identifier `boolean` is a standard sort in the language. The objects of this sort are `false` and `true`.)

10.1.3.4 Constant Declarations

A constant declaration describes expressions of certain syntactic forms as fluent constants or action constants. For a fluent constant, the declaration also specifies its domain. For instance, the declaration

```
:- constants
  loc(block)          :: simpleFluent(location);
  above(block,block)  :: sdFluent;
  move(block,location)   :: action.
```

expresses that the identifier `loc` followed by an object of sort `block` enclosed in parentheses is a regular fluent constant whose domain is the set of locations;

the identifier `above` followed by a pair of objects of sort `block` enclosed in parentheses is a boolean statically determined fluent constant; and the identifier `move` followed by a pair of objects of sorts `block` and `location` enclosed in parentheses is an action constant.

In a constant declaration, the reserved word `inertialFluent` can be used in place of `simpleFluent`. The use of this symbol indicates that a dynamic law expressing the common sense law of inertia for this fluent will be included in translation of the domain description into $\mathcal{BC}$. The reserved word `rigid` has the same meaning as `sdFluent`, except that the syntax of CPLUS2ASP allows a statically determined fluent to be declared as rigid only when its values do not depend on the state. The use of `rigid` allows CPLUS2ASP to process domain descriptions more efficiently.

10.1.4 Atomic Formulas

Atomic formulas are formed from constants, objects, and variables. For instance,

```
loc(B1)=L1,  above(box(1),box(2))=false,  loc(box(1))=table
```

are atomic formulas. The second formula can be also written as

```
-above(box(1),box(2)).
```

10.1.5 Rules

The syntax of rules is the same as the syntax of static and dynamic laws in $\mathcal{BC}$ (Chapter 6), with atomic formulas used as atoms. Abbreviations defined in Sections 6.1, 6.2, 6.4 are also allowed. A rule is followed by a period:

```
intower(B) if loc(B)=table.
move(B,L) causes loc(B)=L.
```

10.1.6 Domain Descriptions

A domain description consists of declarations, rules, and comments, which may appear in any order. For instance, a domain description representing the blocks world in the input language of CPLUS2ASP is shown in Figure 10.1. The translation of that domain description into $\mathcal{BC}$, formed as defined in the next section, is essentially identical to the representation of the blocks world shown in Section 6.5.

10.2 Translation to $\mathcal{BC}$

To specify an action description in language $\mathcal{BC}$, we need to specify its fluent constants along with their domains, its action constants, and its static and dynamic laws (Section 6.1).

```
:- sorts
loc >> block.

:- objects
b1, b2, b3, b4 ::  block;
table ::  loc.

:- constants
loc(block) ::  inertialFluent(loc);
intower(block) ::  sdFluent;
move(block, loc)::  action.

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

% location
impossible loc(B1) = B, loc(B2) = B, B1/=B2.

% Definition of a tower
default -in_tower(B).
intower(B) if loc(B) = table.
intower(B) if loc(B) = B1, intower(B1).

% Blocks don't float in the air
impossible -intower(B).

% Moving a block
move(B,L) causes loc(B)=L.
nonexecutable move(B,L) if loc(B1) = B.
```

Figure 10.1: Blocks World in the language of CPLUS2ASP

10.2.1 Fluent and Action Constants

A sort identifier S represents a set of objects declared as members of the sort, either directly or as members of subsorts of S. For instance, given the sort declarations and the object declarations in the examples of Sections 10.1.3.1

and 10.1.3.2 above, the set associated with the sort `location` consists of the expressions

```
box(1), ..., box(10), table.
```

The set of regular fluent constants is defined by the constant declarations of the forms

$$f(S_1, \dots, S_n) :: \texttt{simpleFluent}(S_{n+1})$$

and

$$f(S_1, \dots, S_n) :: \texttt{inertialFluent}(S_{n+1}).$$

It consists of the symbols $f_{o_1 \dots o_n}$ where each o_i belongs to the set represented by the sort S_i $(i = 1, \dots, n)$. The domain of this fluent constant is the set represented by S_{n+1}.

The statically determined fluent constants and their domains are defined in a similar way by the constant declarations of the forms

$$f(S_1, \dots, S_n) :: \texttt{sdFluent}(S_{n+1})$$

and

$$f(S_1, \dots, S_n) :: \texttt{rigid}(S_{n+1}).$$

The set of action constants is defined by the constant declarations of the form

$$f(S_1, \dots, S_n) :: \texttt{action}.$$

It consists of the symbols $f_{o_1 \dots o_n}$ where each o_i belongs to the set represented by the sort S_i $(i = 1, \dots, n)$.

For instance, in the translation of the action description in Figure 10.1 into $\mathcal{BC}$

- the regular fluent constants are $loc(b_i)$, where $i = 1, 2, 3, 4$; their domain is $\{b_1, b_2, b_3, b_4, table\}$,
- the statically determined boolean fluent constants are $intower(b_i)$, where $i = 1, 2, 3, 4$,
- the action constants are $move(b_i, b_j)$, $move(b_i, table)$, where $i, j = 1, 2, 3, 4$.

10.2.2 Static and Dynamic Laws

The static and dynamic laws of the translation are obtained from rules by grounding. The process of grounding consists of two steps. First, objects of appropriate sorts are substituted for all variables. Second, each expression of the form $o_1 = o_2$ where o_1 and o_2 are objects, is replaced by `true` if o_1 equals o_2, and by `false` otherwise, and similarly for expressions of the form $o_1/ =o_2$.

For instance, the rule

```
intower(B) if loc(B) = table.
```

in Figure 10.1 turns into the set of 4 static laws:

$$\begin{gathered} \mathit{intower}(b_1) \textbf{ if } \mathit{loc}(b_1) = \mathit{table}, \\ \dots \\ \mathit{intower}(b_4) \textbf{ if } \mathit{loc}(b_4) = \mathit{table}. \end{gathered}$$

The dynamic law

```
move(B,L) causes loc(B)=L
```

turns into a set of 20 dynamic laws, such as

$$\mathit{move}(b_1, b_3) \textbf{ causes } \mathit{loc}(b_1) = b_3.$$

which is shorthand for

$$\mathit{loc}(b_1) = b_3 \textbf{ after } \mathit{move}(b_1, b_3) \textbf{ ifcons } \top$$

(Section 6.1).

In addition to the static and dynamic laws obtained by translating rules as shown above, the $\mathcal{BC}$ domain description contains the dynamic laws

$$\textbf{inertial } f_{o_1 \dots o_n}$$

for all regular fluent constants $f_{o_1 \dots o_n}$ corresponding to the declarations of the form

$$f(S_1, \dots, S_n) :: \texttt{inertialFluent}(S_{n+1}).$$

For instance, the translation of Figure 10.1 contains the dynamic laws

$$\begin{gathered} \textbf{inertial } \mathit{loc}(b_1), \\ \dots \\ \textbf{inertial } \mathit{loc}(b_4). \end{gathered}$$

Chapter 11

Reactive Control System in Space Shuttle

Answer set programming has been used to formalize the Reaction Control System (RCS) of space shuttle [65], leading to the USA/RCS-Advisor system that provides decision support for controlling space shuttle maneuvers. The primary responsibility of the RCS system is for maneuvering the space shuttle in space. It consists of fuel and oxidizer tanks, valves and other plumbing needed to provide propellant to the maneuvering jets of the shuttle. It also includes electronic circuitry: both to control valves in the fuel lines and to prepare the jets to receive firing commands. When an orbit maneuver is required, the astronauts must perform actions necessary to prepare the RCS—to close or open valves or energize the proper circuitry. During normal shuttle operations, such plans are pre-scripted so that astronauts know what actions to perform to achieve certain maundering goals. However, since the space shuttle system is complex and involves a high number of components, multiple failures may occur: switches and valves can be stuck in various positions, electrical circuits can malfunction in various ways, valves can be leaking, jets can be damaged, etc. In this situation, the number of possible sets of failures is too large to pre-plan for all of them. It is also not trivial to manually find such a plan, pressured by strict requirement on time and precision, and the cost of a single error can vary from abortion of the mission, in the best scenario, to loss of space vehicles and the crew's lives, in the worst case. The most difficult part is proving that the plan will achieve expected results, given the current condition of the shuttle, without causing any possible dangerous side effects. The USA/RCS-Advisor system serves as a tool to facilitate this task: it can efficiently verify and generate plans that prepare the RCS system for required maneuvers.

The RCS system is one of those dynamic systems that can be described in action description languages. At the time when the USA/RCS-Advisor was created, action languages that were sufficiently expressive for that purpose had been neither carefully defined nor implemented. Its design was inspired earlier work of action description languages, but its implementation was done directly in answer set programming. The code was highly optimized, so that a plan could be efficiently verified or found using the early answer set grounder LPARSE and solver SMODELS. A set of policies— heuristics to shrink search space and speed up plan generation—was included in the RCS system formalization. However, after many years of development of answer set solvers, the same set of plans can be now generated quickly without using any heuristics.

This chapter is about the part of the USA/RCS-Advisor that describes the fluid control system[1]. We reimplemented that piece of software in the fragment of the input language of CPLUS2ASP defined in Chapter 10. Our code

[1]`http://mbal.tk/RCS-ASP/rcs/rcs1`

can be found in Appendix 2. This reformulation shows that the expressiveness of $\mathcal{BC}$ meets the requirements of such a large, complex application. We tested the new implementation using CPLUS2ASP 2, GRINGO 3.0, and CLASP 4.0. In all instances that we experimented with, the new implementation produced exactly the same results as the USA/RCS-Advisor. Since our reformulation uses a higher-level language, it is more compact and user-friendly. As could be expected, on the other hand, it is much less efficient than the original code written in the input language of LPARSE: in our experiments, the solving time is greater by up to 2 orders of magnitude.

11.1 Fluid Control System of the RCS

The RCS system is divided into left RCS, right RCS and forward RCS. Forward RCS is located on the forward fuselage nose area of the orbiter, and left and right RCS are located in the aft fuselage of the orbiter. There are twelve possible maneuvers to be performed by firing jets of the shuttle: +X, -X, +Y, -Y, +Z, -Z, +roll, -roll, +pitch, -pitch, +yaw, -yaw.

The fluid control system of the RCS includes a hydraulic module and a valve control module. The hydraulic module consists of a collection of tanks, jets and pipe junctions connected through pipes. The flow of fluids through the pipe is controlled by valves. The system's purpose is to deliver fuel and oxidizers to the jets needed to perform a maneuver. The structure of the hydraulic system can be represented as a directed graph, with vertices corresponding tanks, jets and pipe junctions, and with arcs labeled by valves. The possible faults of the system at this level include leaky valves, damaged jets, and valves stuck in some position.

Formalizing the hydraulic module involves describing how faults and changes in the position of valves affect the pressure of tanks, jets and junctions. In particular, when fuel and oxidizer flow from the tanks to a properly working jet at the right pressure, the jet is considered ready to fire. In order for a maneuver to start, all the jets it requires must be ready to fire. Pressurization of fuel and oxidizer tanks is obtained by releasing helium from helium tanks connected to the fuel and oxidizer tanks. The necessary condition for a fluid to flow from a tank to a jet, and in general to any node of the directed graph, is that there exists a path without leaks from the tank to the nodes and that all valves along the path are open.

The flow of fuel and oxidizer propellants from tanks to jets is controlled by opening/closing valves along the path connecting these nodes. The state of valves can be changed either by manipulating mechanical switches or by issuing computer commands. Switches and computer commands are connected to the valves they control by electrical circuits.

In some specific phases of operation of the shuttle, such as launching and landing, the on-board general purpose computers, GPCs, will be in charge of opening/closing valves and will achieve this objective by sending computer commands. If the shuttle is in orbit, or the computer system is malfunctioning, an astronaut can normally override these commands by manually flipping the switches that control the valves to be opened or closed. The valve control module describes how computer commands and changes in the position of switches affect the state of valves. The action of flipping a switch to some position normally puts a valve controlled by the switch in that position, and similarly for computer commands. However, switches and valves can be stuck at some position, and electrical circuits can malfunction in various ways. In the formalization investigated in this chapter, we assumed that all electrical circuits are working properly.

11.2 Comparison of Encodings

In this section we compare our encoding (Appendix 2) with the original ASP encoding (`http://mbal.tk/RCS-ASP/rcs/rcs1`). As an example, consider the multi-valued inertial fluent `in_state(Dev)`, which describes the state of the device `Dev`. In the CPLUS2ASP encoding, it is declared as follows:

```
in_state(device):: inertialFluent(state);
```

In the ASP encoding, three rules correspond to this declaration:

```
h(in_state(D,S),T1) :- next(T,T1), of_type(D,Dev), state_of(S,Dev),
                       h(in_state(D,S),T), not -in_state(D,S,T1).

nh(in_state(D,S),T) :- time(T), of_type(D,Dev), state_of(S,Dev),
                  state_of(S1,Dev), neq(S,S1), h(in_state(D,S1),T).
                  :- time(T), of_type(D,Dev), state_of(S,Dev),
                     h(in_state(D,S),T), nh(in_state(D,S),T).
```

The CPLUS2ASP formulation is more compact because the idea of inertia and the uniqueness of the value of a multi-valued fluent are incorporated in the syntax and semantics of $\mathcal{BC}$, and because action description languages do not use time stamps.

The need to include the atoms `of_type(D,Dev)` and `state_of(S,Dev)` in the bodies of the rules is explained by the fact that there are no variable declarations in traditional ASP languages. The new ASP language $\mathcal{SPARC}$ [3] addresses this problem by allowing the user to include typing information in the ASP code. For instance, the predicate `in_state` can be declared as follows:

```
in_state(#device,#state,#time).
```

In the presence of this declaration, the three rules shown above can be written more concisely:

```
in_state(D,S,T1) :- next(T,T1), in_state(D,S,T),
                    not -in_state(D,S,T1).


-in_state(D,S,T) :- neq(S,S1), in_state(D,S1,T).


:- h(in_state(D,S),T), nh(in_state(D,S),T).
```

11.3 Operation of CPLUS2ASP

The system CPLUS2ASP translates action descriptions in its input language into logic programming rules. This process is somewhat similar, but not quite identical to the composition of two translations defined in this dissertation: the translation from CPLUS2ASP into $\mathcal{BC}$ (Section 10.2), and the translation from $\mathcal{BC}$ description into logic programming (Section 6.3). The main difference is that it produces a program with variables that needs to be grounded.

Consider, for instance, the rule

```
flip(Sw,S) causes in_state(Sw)=S if -stuckd(Sw).        (11.1)
```

The translation into $\mathcal{BC}$ defined in Section 10.2 turns it into the set of dynamic laws

$$\mathit{flip}_{o_1,o_2}\ \textbf{causes}\ \mathit{in\_state}(o_1) = o_2\ \textbf{if}\ \sim \mathit{stuckd}(o_1)$$

for all members o_1 of sort `switch` and all members o_2 of sort `state`. The translation PN_l further converts these dynamic laws into the rules

$$i+1 : \mathit{in\_state}(o_1) = o_2 \leftarrow i : \mathit{flip}_{o_1,o_2} \wedge i : \sim \mathit{stuckd}(o_1) \tag{11.2}$$

where $0 \leq i \leq l-1$.

On the other hand, CPLUS2ASP translates (11.1) into a logic programming rule with variables:

```
h(eql(in_state(Sw),S),V_astep) :- h(eql(stuckd(Sw),false)),
                                  occ(eql(flip(Sw,S),true),V_astep-1).
```

Here, `h` stands for "holds," `eql` stands for "equals," and `occ` stands for " occurs."

11.4 Plan Generation

To generate a plan, the initial conditions and the goal of the planning query should be added to the output of CPLUS2ASP, and the resulting program should be sent to an answer set solver.

The rules in Figure 11.2 correspond to the query

`http://mbal.tk/RCS-ASP/rcs/instances/instances-manual/instance_003`

It describes the initial state where the tanks that contain fuel and oxidizers are pressurized properly, there is no abnormal input to the valve and all other nodes are not pressurized. All switches are in the `gpc` mode except for the switch `lx12`, which is `closed`. Some valves are closed and others are open. On the path to the jets, some valves have leaks and some switches are stuck. The planning goal is specified by three constraints.

The first stable model generated by CLINGO with `maxstep=7` is shown in Figure 11.1.

```
occ(eql(flip(lx345,gpc),true),6)
occ(eql(flip(fhb,open),true),6)
occ(eql(issue(open_loi345a),true),6)
occ(eql(flip(fi345,open),true),5)
occ(eql(issue(open_rfm4),true),5)
occ(eql(flip(fi12,open),true),4)
occ(eql(flip(rx12,open),true),4)
occ(eql(issue(closeb_lfm1),true),4)
occ(eql(flip(fm1,open),true),3)
occ(eql(flip(li12,open),true),3)
occ(eql(issue(openb_roi12),true),3)
occ(eql(flip(fm4,open),true),2)
occ(eql(flip(rx345,open),true),2)
occ(eql(flip(lx345,open),true),2)
occ(eql(flip(fm3,closed),true),1)
occ(eql(issue(opena_rfha),true),1)
occ(eql(issue(opena_lfhb),true),1)
occ(eql(flip(fm2,closed),true),0)
occ(eql(flip(lm4,open),true),0)
occ(eql(issue(openb_rfx12),true),0)
```

Figure 11.1: The plan for the maneuvering task

For two smaller planning problems, we verified that the set of plans that can be generated using our reimplementation of the USA/RCS-Advisor is the same as the set of plans generated by the original system. For one problem, the number of plans is around 5000, and the other is close to 100,000.

The table below compares the grounding time, solving time, and the sizes of the grounded program for the original ASP encoding and for the new implementation. All numbers for the new implementation are larger by 2-3 orders of magnitude. This fact shows that the translation implemented in the current version of CPLUS2ASP needs to be significantly improved before the system becomes applicable to industrial-size domains.

	$\mathcal{BC}$ *encoding*	ASP encoding
grounding time(sec)	28.179	0.056
solving time(sec)	5.281	0.03
ground file size	118M	140K

In this chapter we showed how to use a subset of the input language of CPLUS2ASP for formalizing the valve control system and hydraulic system of the RCS. Compared to the original formulation, the input to CPLUS2ASP is more concise and easier to understand, but significantly less efficient.

```
% initial state

h(eql(pressurized_by(ffh,ffh),true),0).
h(eql(pressurized_by(foh,foh),true),0).
h(eql(pressurized_by(lfh,lfh),true),0).
h(eql(pressurized_by(loh,loh),true),0).
h(eql(pressurized_by(rfh,rfh),true),0).
h(eql(pressurized_by(roh,roh),true),0).
h(eql(ab_input(V),false),0).
h(eql(pressurized_by(N,T),false),0) :- N!=T.
h(eql(pressurized_by(T,T),false),0) :- T!=ffh, T!=foh,
                         T!=lfh, T!=loh, T!=rfh, T!=roh.
h(eql(in_state(Swi),gpc),0) :- Swi!=lx12.
h(eql(in_state(V),closed),0):- V!=ffdummy, V!=fodummy,
                                V!=lfdummy, V!=lodummy,
                                V!=rfdummy, V!=rodummy,
                                V!=ffm2, V!=fom3,
                                V!=lfx345, V!=lom1,
                                V!=rfm3,V!=rfm4,
                                V!=rfi345a, V!=roi345b,
                                V!=rfi12.
h(eql(in_state(DV),open),0).
h(eql(has_leak(ffm2),true)).h(eql(has_leak(fom3),true)).
h(eql(has_leak(rfm3),true)).h(eql(has_leak(rfm4),true)).
h(eql(has_leak(lom1),true)).h(eql(has_leak(lfx345),true)).
h(eql(stuck(rfi345a,closed),true)).
h(eql(stuck(roi345b,closed),true)).
h(eql(stuck(rfi12,closed),true)).h(eql(stuck(lx12,closed),true)).
h(eql(in_state(ffm2),open),0).h(eql(in_state(fom3),open),0).
h(eql(in_state(lfx345),open),0).h(eql(in_state(lom1),open),0).
h(eql(in_state(rfm3),open),0).h(eql(in_state(rfm4),open),0).

%goal

false :- not h(eql(maneuver_of(minus_z,left_rcs),true),maxstep).
false :- not h(eql(maneuver_of(minus_z,fwd_rcs),true),maxstep).
false :- not h(eql(maneuver_of(minus_z,right_rcs),true),maxstep).
```

Figure 11.2: The planning query for a maneuvering task

Chapter 12

Task Planning for Mobile Robots

As robots deal with increasingly complex tasks, automated planning systems can provide great flexibility over direct implementation of behaviors for robotic tasks. In mobile robotics, uncertainty about the environment stems from many sources, which is particularly true for domains inhabited by humans, where the state of the environment can change outside the robot's control in ways that are difficult to predict.

The qualitative modeling of dynamic domains at a given abstraction level, based on a formal language, allows for the generation of provably correct plans. The brittleness owing to the prevalent uncertainty in the model can be overcome through execution monitoring and replanning, when the outcome of an action deviates from the expected effect.

In this chapter we demonstrate that the action language $\mathcal{BC}$ can be used for robot task planning in realistic domains that require planning in the presence of missing information and indirect action effects. These features are necessary to completely describe many complex tasks. For instance, in a task where a robot has to collect mail intended for delivery from all building residents, the robot may need to visit a person whose location it does not know.

To overcome this problem, it can plan to complete its task by asking someone else for that person's location, thereby acquiring this missing information. Additionally, a person may forward his mail to another person in case he will be unavailable when the robot comes around to collect mail. In such situations, the information about mail transfers is best expressed through a recursive definition. When the robot visits a person who has mail from multiple people, planning needs to account for the fact that mail from all these people will be collected indirectly. In this chapter, we use this mail collection task to demonstrate how these problems can be solved. The overall methodology is applicable to other planning domains that involve recursive fluents, indirect action effects, and human-robot interaction.

Furthermore, we show in this chapter how answer set planning under action costs [12] can be applied to robot task planning. Incorporating costs in symbolic planning is important for applications that involve physical systems and deal with limited resources such as time, battery, communication bandwidth, etc. Previous applications of action languages to robotics [7, 9] do not consider these costs. A robot can learn these costs from experience [37, 73].

12.1 Architecture Description

Figure 12.1 shows an architecture that integrates deterministic planning, execution monitoring, replanning and cost learning. It includes three modules: a planner and a cost learner. At planning time, the planner generates a description of the initial state of the world based on observations

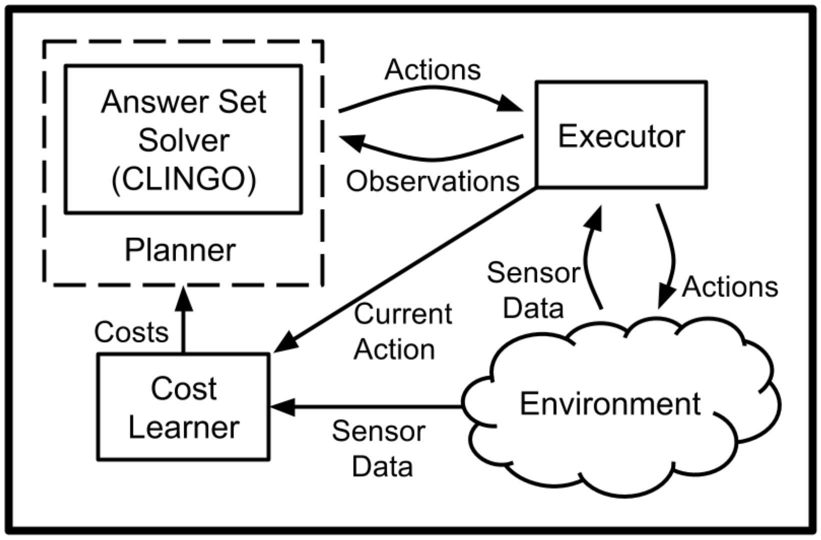

Figure 12.1: The architecture used in our approach. The planner invokes a cost learner that learns costs from sensing during execution.

provided by the executor. The initial state, the goal provided by the user and the domain description (translated by CPLUS2ASP are sent to an answer set solver CLINGO. CLINGO polls the cost of each action from the cost learner, and produces an optimal plan. After plan generation, the executor invokes the appropriate controllers for each action, grounds numeric sensor observations into information about values of fluents, and returns these information to the planning module. If the observed values are incompatible with the state expected by the planner, then the planner will update the robot's domain knowledge and replan. During action execution, the cost learner employs a learning algorithm to estimate the cost of each action from the experienced samples.

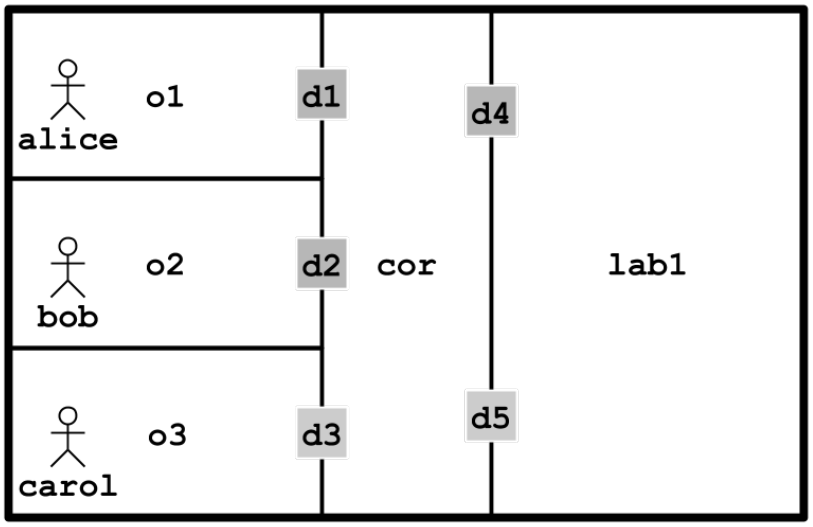

Figure 12.2: The layout of the example floor plan used in the text along with the information about the residents of the building.

12.2 Domain Representation

In order to demonstrate how ASP can be used for robot task planning under incomplete information, with human-robot interaction and with action costs, we use a small domain as a running example. The example domain we consider has a mobile robot that navigates inside a building, visiting and serving the inhabitants by collecting mail:

> The robot drops by offices at 2pm every day to collect outgoing mail from the residents. However, some people may not be in their offices at that time, so they can pass their outgoing mail to colleagues in other offices, and send this information to the robot. When the robot collects the mail, it should obtain it while only

visiting people as necessary. If the robot needs to collect mail from a person whose location is not known, it should plan to visit other people to acquire this information.

The floor plan of the example building is shown in Figure 12.2. In this example, we consider the following objects:

- *alice*, *bob*, *carol* and *dan* are people,
- o_1, o_2, o_3 are offices and lab_1 is a lab,
- *cor* (corridor) is a room,
- d_1, d_2 and d_3 are doors.

In the following subsections, we will use meta-variables $P, P_1, P_2, \ldots$ to denote people, $R, R_1, R_2, \ldots$ to denote rooms, offices and labs, and $D, D_1, D_2, \ldots$ to denote doors.

Domain knowledge about a building includes three types of information: *rigid knowledge*, *time-dependent knowledge*, and *action knowledge*. We explain each of these in detail in the following subsections.

12.2.1 Rigid Knowledge

Rigid knowledge includes information about the building that does not depend upon the passage of time. In our example, rigid knowledge includes accessibility between the rooms, the lab, and the corridor. This knowledge has been formalized as follows:

- $hasdoor(R, D)$: office R has door D. An office R does not have a door D unless specified. This default expresses the closed world assumption [67] for $hasdoor$:

  $hasdoor(o_1, d_1)$ $hasdoor(o_2, d_2)$ $hasdoor(o_3, d_3)$
  $hasdoor(lab_1, d_4)$ $hasdoor(lab_1, d_5)$
  default $\sim hasdoor(R, D)$.

- $acc(R_1, D, R_2)$: room R_1 is accessible from room R_2 via door D. Two rooms are not connected by a door unless specified:

  $acc(R, D, cor)$ **if** $hasdoor(R, D)$
  $acc(R, D, cor)$ **if** $acc(cor, D, R)$
  default $\sim acc(R_1, D, R_2)$.

- $knows(P_1, P_2)$ describes P_1 knows where person P_2 is. By default, a person P_1 does not know where another person P_2 is.

- $passto(P_1, P_2)$: person P_1 has passed mail to person P_2. By default, a person P_1 has not passed mail to a person P_2 (including himself).

12.2.2 Time-Dependent Knowledge

Time-Dependent Knowledge includes information about the environment that can change with the passage of time, as the robot moves around in the environment. It has been formalized as follows:

- The current location of a person is expressed by the fluent $inside$. $inside(P, R)$ means that person P is located in room R. A person can only be inside a single room at any given time. The fluent is inertial:

  $\sim inside(P, R_2)$ **if** $inside(P, R_1)$ $(R_1 \neq R_2)$
  inertial $inside(P, R)$.

- Whether the robot knows the current location of a person is expressed by the fluent *knowinside*. $knowinside(P, R)$ means that the robot knows that person P is located in room R. The robot knows that a person can only be inside a single room at any given time. The fluent is inertial:

  $\sim knowinside(P, R_2)$ **if** $knowinside(P, R_1)$ $(R1 \neq R_2)$
  inertial $knowinside(P, R)$.

  If the robot knows that a person P is in room R, then P is indeed in room R:

  $$inside(P, R) \textbf{ if } knowinside(P, R).$$

- $open(D)$: a door D is open. By default, a door is not open.

- $visiting(P)$: the robot is visiting a person P. By default, a robot is not visiting anyone.

- The fluent $mailcollected(P)$ means the robot has collected mail from P. This fluent is inertial. It is recursively defined as follows. The robot has collected P_1's mail if it has collected P_2's mail and P_1 has passed his mail to P_2.

  $$mailcollected(P_1) \textbf{ if } mailcollected(P_2), passto(P_1, P_2). \tag{12.1}$$

- $facing(D)$: the robot is next to a door D and is facing it. The robot cannot face two different doors simultaneously.

- $beside(D)$: the robot is next to door D. $beside(D)$ is true if *facing(D)* is true, and the robot cannot be beside two different doors simultaneously.

Since *beside* is implied by *facing*, it will become an indirect effect of the actions that make the fluent *facing* true.

- $loc = R$: the robot is at room R.

12.2.3 Action Knowledge

Action knowledge includes the rules that formalize the actions of the robot, the preconditions for executing those actions, and the effects of those actions. The robot can execute the following actions:

- $approach(D)$: the robot approaches door D. The robot can only approach a door accessible from the the robot's current location and if it is not facing the door. Approaching a door causes the robot to face that door.

  $approach(D)$ **causes** $facing(D)$
  nonexecutable $approach(D)$ **if** $loc = R, \sim hasdoor(R, D)$
  nonexecutable $approach(D)$ **if** $facing(D)$.

- $gothrough(D)$: the robot goes through door D. The robot can only go through a door if the door is accessible from the robot's current location, if it is open, and if the robot is facing it. Executing the *gothrough* action results in the robot's location being changed to the connecting room and the robot no longer facing the door.

- $greet(P)$: the robot greets person P. A robot can only greet a person if the robot knows that both the robot and that person are in the same room. Greeting a person P results in the $visiting(P)$ fluent being true.

- *collectmail*(P): the robot collects mail from person P. A robot can only collect mail from a person if the robot knows that both the robot and that person are in the same room, if the person has not passed his mail to someone else, and if the person's mail has not been collected yet. Collecting mail from a person P results in the *mailcollected*(P) fluent being true, formalized as

  $$\textit{collectmail}(P) \textbf{ causes } \textit{mailcollected}(P)$$

  Because of the recursive definition of *mailcollected* in (12.1), *collectmail*(P) will also *indirectly* lead to the other people's mail passed to P to be collected as well.

- *opendoor*(D): the robot opens a closed door D. The robot can only open a door that it is facing.

- *askploc*(P): The robot asks the location of person P if it does not know the location of person P. Furthermore, the robot can only execute this action if it is visiting a person P_1 who knows the location of person P. This is the action that triggers human-robot interaction. By executing this action, the robot knows that the location of person P is room R, formalized as

  $$\textit{askploc}(P_1, P) \textbf{ causes } \textit{knowinside}(P, R) \textbf{ if } \textit{inside}(P, R).$$

A complete description of this domain in the fragment of the input language of CPLUS2ASP defined in Chapter 10 is shown in Appendix 3.

12.3 Planning for a Mobile Robot

12.3.1 Generating and executing plans

Before a plan can be generated, the planner needs to obtain an initial state from two sources:

- The planner maintains tables for some portion of the domain knowledge, namely, *knowinside*, *knows*, and *passto*, that help the robot reason about acquiring missing information and figure out how mail has been forwarded recursively. At planning time, the contents of the table are translated into a part of query that describes the initial state. For instance, the table that contains fluent values for *knowinside* is:

knowinside	o_1	o_2	o_3	lab_1
alice	**t**	**f**	**f**	**f**
bob	**f**	**t**	**f**	**f**
carol	**f**	**f**	**t**	**f**
dan	**f**	**f**	**f**	**f**

(12.2)

  Using this table, the planner outputs the table contents as a set of atoms which are joined to the query:

$$\begin{array}{l} knowinside(alice, o_1), \sim knowinside(alice, o_2), \ldots, \\ \sim knowinside(bob, o_1), knowinside(bob, o_2), \ldots, \end{array}$$

  Note that all values in the last row of the table are **f**, indicating that Dan's location is not known.

- The planner polls the sensors to obtain the values of some portion of the time-dependent knowledge, namely, *beside*, *facing*, *open* and *loc*, and translates them into a part of the query that describes the initial state. The sensors guarantee that the value for *loc* is always returned for exactly one location, and *beside* and *facing* are returned with at most one door. If the robot is facing a door, the value of *open* for that door is sensed and returned as well. For instance, in the initial state, if the robot is in lab_1 and not facing any door, the planner senses and appends the following to the description of the initial state:

$$loc = lab_1, {\sim}beside(d_4), {\sim}facing(d_4), \ldots$$

In addition to the initial state, the query includes also a goal, for instance, *visiting*(*alice*).

To find the shortest plan, CLINGO is called repeatedly with larger and larger plan lengths up to a user-defined number *maximumLength*. Execution is stopped at the first length for which a plan exists. In the case where the robot starts in lab_1 and its goal is *visiting*(*alice*), the following 7-step is generated by CLINGO:

$$\begin{array}{l} 0\!:\!approach(d_5), 1\!:\!opendoor(d_5), 2\!:\!gothrough(d_5), \\ 3\!:\!approach(d_1), 4\!:\!opendoor(d_1), 5\!:\!gothrough(d_1), \\ 6\!:\!greet(alice) \end{array}$$

The output of CLINGO also contains the values of the fluents at various times:

$$0\!:\!loc = lab_1, 0\!:\!{\sim}facing(d_5), 1\!:\!loc = lab_1, 1\!:\!facing(d_5), \ldots$$

These values are used to monitor execution. For instance, assume that the robot is executing $approach(d_5)$ at time 0. The robot attempts to navigate to door d_5, but fails and returns an observation $\sim facing(d_5)$ at time 1. Since this observation does not match the expected effect of $approach(d_5)$ at time 1, which is $facing(d_5)$, the robot incorporates $\sim facing(d_5)$ as part of a new initial condition and plans again.

12.3.2 The Mail Collection Task

To solve the mail collection problem, the robot first needs to receive information about how mail was transferred from one person to another person, i.e., information that relates to the fluent *passto*. Any person who passes his mail to other people will send this information to the robot.

In our example domain, let's assume that the robot receives the following information:

passto	*alice*	*bob*	*carol*	*dan*
alice	**f**	**f**	**f**	**f**
bob	**t**	**f**	**f**	**f**
carol	**f**	**f**	**f**	**f**
dan	**f**	**t**	**f**	**f**

(12.3)

Let's assume that initially the robot is in lab_1 and not beside nor facing any door. The goal of collecting everyone's mail and reaching the corridor can be described as:

$$\begin{array}{l} mailcollected(alice), mailcollected(bob), \\ mailcollected(carol), mailcollected(dan), loc = cor. \end{array}$$

CLINGO generates an answer set with the following plan:

$$\begin{array}{l}
0\!:\!\mathit{approach}(d_5), 1\!:\!\mathit{opendoor}(d_5), 2\!:\!\mathit{gothrough}(d_5),\\
3\!:\!\mathit{approach}(d_1), 4\!:\!\mathit{opendoor}(d_1), 5\!:\!\mathit{gothrough}(d_1),\\
6\!:\!\mathit{collectmail}(\mathit{alice}),\\
7\!:\!\mathit{approach}(d_1), 8\!:\!\mathit{opendoor}(d_1), 9\!:\!\mathit{gothrough}(d_1),\\
10\!:\!\mathit{approach}(d_3), 11\!:\!\mathit{opendoor}(d_3), 12\!:\!\mathit{gothrough}(d_3),\\
13\!:\!\mathit{collectmail}(\mathit{carol}),\\
14\!:\!\mathit{approach}(d_3), 15\!:\!\mathit{opendoor}(d_3), 16\!:\!\mathit{gothrough}(d_3)
\end{array}$$

In this plan, the robot only visits Alice and Carol, and doing so is sufficient to collect everyone's mail, even if Dan's location is not known.

12.3.3 Planning with Human Robot Interaction

Consider the modification of Table 12.3 in which Dan doesn't forward his mail to Bob. To collect Dan's mail, the robot now needs to visit him. However, the robot does not know where Dan is, as shown in the last row of Table 12.2. In our example domain, we assume Carol knows Dan's location:

knows	*alice*	*bob*	*carol*	*dan*
alice	**f**	**f**	**f**	**f**
bob	**f**	**f**	**f**	**f**
carol	**f**	**f**	**f**	**t**
dan	**f**	**f**	**f**	**f**

Again, let's assume that the robot is initially located in lab_1 and not beside nor facing any door. The planner calls CLINGO with the same initial state and the same goal as in the previous section to generate the following shortest plan:

$$\begin{array}{l} 0\!:\!\mathit{approach}(d_5), 1\!:\!\mathit{opendoor}(d_5), 2\!:\!\mathit{gothrough}(d_5), \\ 3\!:\!\mathit{approach}(d_1), 4\!:\!\mathit{opendoor}(d_1), 5\!:\!\mathit{gothrough}(d_1), \\ 6\!:\!\mathit{collectmail}(\mathit{alice}), \\ 7\!:\!\mathit{approach}(d_1), 8\!:\!\mathit{opendoor}(d_1), 9\!:\!\mathit{gothrough}(d_1), \\ 10\!:\!\mathit{approach}(d_3), 11\!:\!\mathit{opendoor}(d_3), 12\!:\!\mathit{gothrough}(d_3), \\ 13\!:\!\mathit{collectmail}(\mathit{carol}), \\ 14\!:\!\mathit{greet}(\mathit{carol}), 15\!:\!\mathit{askploc}(\mathit{dan}), 16\!:\!\mathit{collectmail}(\mathit{dan}), \\ 17\!:\!\mathit{approach}(d_3), 18\!:\!\mathit{opendoor}(d_3), 19\!:\!\mathit{gothrough}(d_3) \end{array}$$

The first 13 steps of this plan are the same as in the plan shown in Section 12.3.2. It is important to notice that the answer set also contains the following atom:

$$16\!:\!\mathit{knowinside}(\mathit{dan}, o_3) \tag{12.4}$$

This atom describes the effect of executing action *askploc*(*dan*) at time 15. Since CLINGO searches for the shortest plan by incrementing the number of steps, the "optimistic" plan that it finds corresponds to the case where Dan is located at the same office as Carol.

As before, the plan is executed and the execution is monitored. The robot executes action *askploc*(*dan*) at time 15 by asking Carol for Dan's location. The robot obtains Carol's answer as an atom, for instance,

$$16\!:\!\mathit{knowinside}(\mathit{dan}, o_2),$$

which contradicts (12.4). As in the case of execution failure, replanning is necessary. Before replanning, the acquired information is used to update Table 12.2. While replanning, the time stamp is reset to start from 0, and the updated table will generate the new initial condition $0\!:\!\mathit{knowinside}(\mathit{dan}, o_2)$.

After running CLINGO again, a new plan is found based on the information acquired from Carol:

$$\begin{array}{l} 0\!:\!\mathit{approach}(d_3), 1\!:\!\mathit{opendoor}(d_3), 2\!:\!\mathit{gothrough}(d_3), \\ 3\!:\!\mathit{approach}(d_2), 4\!:\!\mathit{opendoor}(d_2), 5\!:\!\mathit{gothrough}(d_2), \\ 6\!:\!\mathit{collectmail}(\mathit{dan}), \\ 7\!:\!\mathit{approach}(d_2), 8\!:\!\mathit{opendoor}(d_2), 9\!:\!\mathit{gothrough}(d_2) \end{array}$$

By interacting with Carol, the robot obtained Dan's location, updated its knowledge base, and completed its goal of collecting mail from everyone including Dan.

12.3.4 Planning with Action Costs

In the previous section, the planner generates multiple plans of equal length out of which one is arbitrarily selected for execution. But those plans may not be equivalent to each other in the sense that different actions may have different costs. In our domain, we consider the cost of an action to be the time spent during its execution. For instance, when the robot visits Alice in the first few steps of plan execution, the generated plan includes the robot exiting lab_1 through door d_5. The planner also generated another plan of the same length where the robot could have exited through door d_4, but that plan was not selected. If we look at the layout of the example environment in Figure 12.2 then we can see that it is indeed faster to reach Alice's office o_1 through door d_4. In this section, we show how costs can be associated with actions, and how a plan with the smallest cost can be found.

Costs are functions of both the action being performed and the state at the beginning of that action. CPLUS2ASP does not directly support formalizing costs for generating optimal plans, but CLINGO allows the user to write a logic program with *optimization statements* (indicated via the keywords `#maximize` and `#minimize`) to generate optimal answer sets. Therefore, in our application, cost formalization and optimization statements are directly written in logic program rules in CLINGO syntax. They are then appended to the domain description and query and sent to CLINGO to generate an optimal plan.

In the example domain, we assume that all actions other than *approach* have fixed costs. Specifically, the actions *askploc*, *opendoor*, *greet*, and *collectmail* have cost 1, and *gothrough* has cost 5. The cost of executing the action *approach*(D) depends on the physical location of the robot. It is computed in two different ways:

- When the robot approaches door D_1 from door D_2 and is currently located in R, the values of fluents uniquely identify the physical location of the robot in the environment. The cost of action *approach*(D_1) is specified by the external term `@cost(D1,D2,R)`. At the time of plan generation, CLINGO will make external function calls to compute the value of the external term.

- When the robot is not next to a door (for instance, in the middle of the corridor), the cost for approaching any door is fixed to 10. This only happens in the initial condition.

With costs assigned to actions, we use the optimization statement `#minimize` to guide the solver to return a plan of optimal cost, instead of the shortest plan. When costs are involved, we do not call CLINGO repeatedly while incrementing values of maximum plan length; instead, we directly search for the optimal plan with a maximum of *maximumLength* steps. It is important to note that the optimal plan found by CLINGO is not necessarily the global optimal plan, but only the optimal plan up to *maximumLength* steps. *maximumLength* needs to be set appropriately to balance optimality with execution time based on computational power available.

In this chapter, we introduced an approach that uses action language $\mathcal{BC}$ for robot task planning and incorporates action costs to produce optimal plans. We applied this approach to a mail collection task. Using action language $\mathcal{BC}$ allows us to formalize indirect effects of actions on recursive fluents. In the presence of incomplete information, the proposed approach can generate plans to acquire missing information through human-robot interaction.

The planning method described in this chapter has been used to implement a task planner for robots operating in the 3NE Wing of the GDC building at the University of Texas at Austin, within the framework of the "Building-Wide Intelligence" project led by Professor Peter Stone. The project was also contributed by Piyush Khandelwal and Matteo Leonetti. The source code for that application is shown in Appendix 3.2.

Chapter 13

Conclusion

The dissertation addressed problems of three kinds.

First, we studied some mathematical properties of expressive action languages based on nonmonotonic causal logic that had not been well understood in the past. This includes causal rules expressing synonymy, nondefinite causal rules, and nonpropositional causal rules. We generalized existing translations from nonmonotonic causal theories to logic programming under the answer set semantics. We extended the notion of a definite causal theory by allowing explainable function symbols in the head, and generalized literal completion to such theories. After that, we generalized McCain's translation and Ferraris's translation, which had been originally defined for propositional causal theories, to the first-order case. We also studied translating synonymy rules into logic programs under the first-order stable model semantics. Finally, we addressed the problem of eliminating explainable function symbols from a causal theory in favor of predicate symbols. This makes it possible to automate reasoning with a wider class of causal theories by calling answer set solvers.

Second, we designed and studied a new action language $\mathcal{BC}$, which is more expressive in some ways than the existing and previously proposed languages. We developed a framework that combines the most useful expressive features of the languages $\mathcal{B}$ and $\mathcal{C}+$, and investigated their relationship to $\mathcal{BC}$. Furthermore, we studied the cases where program completion can be used to characterize the effects of actions described in these languages. We generalized the concept of tight logic program by introducing rule-dependency graph.

Third, we illustrated the possibilities of the new action language by two practical applications: to the dynamic domain of the Reactive Control System of the space shuttle, and to the task planning of mobile robots. We began with defining the syntax of a fragment of the input language of software system CPLUS2ASP version 2, and characterized its semantics by a translation into a $\mathcal{BC}$ action description. After that, we used this language to reimplement a large part of the USA/RCS-Advisor. Finally, we used this language to formalize the dynamic domain of a mobile robot, and showed how the domain description can be used to generate optimal plans, monitor execution, replan, and interact with humans to gather information. These applications illustrate the expressiveness of the language $\mathcal{BC}$ and demonstrate its applicability to realistic domains.

The future work of this dissertation may involve the following two aspects. First, action language $\mathcal{BC}$ can be generalized into first-order case where variables are not longer interpreted as meta-variables. Second, we will further investigate more sophisticated application in robotics, involving using hierarchical domain representation to speed up planning, incorporating human-robot dialogue to facilitate planning under incomplete information, and further integrating planning with probablistic models such as partially observeable Markov decision process (POMDP).

Appendix

Appendix 1

Syntax of The Input Language of Chapter 10

The context-free grammar below uses EBNF notation. The symbols

```
::=    |    {    }
```

belong to EBNF notation. The vertical bar expresses disjunction and curly braces expresses grouping. Nonterminals begin with a capital letter. Many of the parentheses in this grammar can be dropped due to operator precedences.

1.1 Identifiers and Integers

```
ExtendedInteger ::= integer | - ExtendedInteger
                  | ( ExtendedInteger { + | - | * | // | mod }
                      ExtendedInteger )

      Num_range ::= ExtendedInteger .. ExtendedInteger
```

The second operand of // (which stands for integer division) and the second operand of mod must have nonzero values.

The expressions based on extended integers are defined as follows.

```
Extended_integer_outer_expression::= Extended_integer_expression
                                  | ( Extended_integer_expression )
```

```
Extended_integer_expression::=  Extended_integer
                             | Extended_integer_outer_expression +
                               Extended_integer_outer_expression
                             | Extended_integer_outer_expression -
                               Extended_integer_outer_expression
                             | Extended_integer_outer_expression *
                               Extended_integer_outer_expression
                             | Extended_integer_outer_expression //
                               Extended_integer_outer_expression
                             | Extended_integer_outer_expression mod
                               Extended_integer_outer_expression
```

1.2 Declarations

1.2.1 Sorts

Sorts are denoted by identifiers, and their declarations have the following syntax:

```
       Sort_statement::=  :- Sorts Sort_spec_outer_tuple .

Sort_spec_outer_tuple::= Sort_spec_tuple
                        | ( Sort_spec_tuple )

      Sort_spec_tuple::= Sort_spec
                        | Sort_spec_outer_tuple ; Sort_spec_outer_tuple
                        | Sort_spec_outer_tuple , Sort_spec_outer_tuple

            Sort_spec::= Sort_identifier_no_range
                        | Sort_identifier_no_range >> Sort_spec_outer_tuple

Sort_identifier_no_range::= Sort_identifier_name
                        | Sort_identifier_name *
                        | Sort_identifier_name + none

    Sort_identifier_name::=  identifier
```

1.2.2 Objects

```
       Object_statement::= :- objects Object_spec_outer_tuple .

Object_spec_outer_tuple::= Object_spec_tuple
                         | ( Object_spec_tuple )

       Object_spec_tuple::= Object_outer_spec
                         | Object_spec_outer_tuple ; Object_spec_outer_tuple

       Object_outer_spec::= Object_spec
                         | ( Object_spec )

             Object_spec::= Object_outer_schema_list :: Sort_outer_identifier

Object_outer_schema_list::= Object_schema_list
                         | ( Object_schema_list )

       Object_schema_list::= Object_outer_schema
                         | Object_outer_schema_list, object_outer_schema

      Object_outer_schema::= Object_schema
                         | ( Object_schema )

            object_schema::= identifier
                         | identifer ( Sort_identifier_list )
                         | Extended_integer_outer_expression
                         | Num_range
```

1.2.3 Variables

Variable identifier must begin with capitalized letter.

```
        Variable_statement::= :- variables Variable_spec_outer_tuple .

 Variable_spec_outer_tuple::= Variable_spec_tuple
                            | ( Variable_spec_tuple )

       Variable_spec_tuple::= Variable_outer_spec
```

```
                   | Variable_outer_spec ;
                     variable_spec_outer_tuple

   Variable_outer_spec::= Variable_spec | ( Variable_spec )

        Variable_spec ::= Variable_outer_list ::
                            Variable_binding

   Variable_outer_list::= Variable_list | ( Variable_list )

         Variable_list::= identifier
                   | Variable_list , identifier

      Variable_binding::= Sort_identifier_no_range
                   | ( Sort_identifier_no_range )
                   | Num_range
```

1.2.4 Constants

```
      Constant_statement::= :- constants Constant_spec_outer_tuple .

Constant_spec_outer_tuple::= Constant_spec_tuple | ( Constant_spec_tuple )

      Constant_spec_tuple::= Constant_outer_spec
                   | Constant_spec_outer_tuple ; Constant_outer_spec

      Constant_outer_spec::= Constant_spec | ( Constant_spec )

            Constant_spec::= Constant_schema_outer_list ::
                             Constant_outer_binder

Constant_schema_outer_list::= Constant_schema_list
                   | ( Constant_schema_list )

      Constant_schema_list::= Constant_outer_schema
                   | Constant_schema_outer_list ,
                     Constant_outer_schema

     Constant_outer_schema::= Constant_schema
```

```
                     | ( Constant_schema )

         Constant_schema::= identifier
                          | identifier ( Sort_identifier_list )

  Constant_schema_nodecl::= identifier
                          | identifier ( Sort_identifier_list )

   Constant_outer_binder::= Constant_binder
                          | ( Constant_binder )

         Constant_binder::= Sort_constant_name ( Sort_identifier )
                          | Num_range
```

1.2.5 Rules

```
     Atom_list ::= Atom | Atom, Atom_list

          Atom ::= identifier | identifier(Argument_list)
                | identifier = identifier
                | identifier(Argument_list) = identifier
                | identifier(Argument_list) = identifier(Argument_list)
                | identifier \= identifier
                | identifier(Argument_list) \= identifier
                | identifier(Argument_list) \= identifier(Argument_list)

 Argument_list ::= identifer| identifier Argument_list

         Rules ::= empty | Static_law | Dynamic_law

    Static_law ::= Atom if Atom_list ifcons Atom_list .
                | Atom if Atom_list .
                | Atom .
                | default Atom .
                | default Atom if Atom_list .
                | impossible Atom_list .

   Dynamic_law ::= Atom after Atom_list ifcons Atom_list .
                | Atom after Atom_list .
```

```
| Atom causes Atom if Atom_list .
| default Atom after Atom_list .
| inertial identifer .
| inertial identifer(Argument_list) .
| nonexecutable Atom_list .
```

1.3 Non-Context-Free Regulations

To write a correct description, the following regulations are imposed.

- The following words are reserved from the set of identifiers:

  sorts, objects, variables, constants, action, rigidFluent, sdFluent, simpleFluent, inertialFluent, if, ifcons, after, causes, default, inertial, nonexecutable, impossible, boolean.

- The set of object declarations in a CCALC input file must be acyclic. For instance,

  ```
  next(index) :: index
  ```

  is not allowed. The acyclicity condition guarantees that the set of objects of each sort is finite.

- An identifier used as a name for an object and a constant may not be declared more than once. If a variable is declared more than once then it should be assigned the same sort in all its declarations.

- `Argument_list` should only contain object identifiers and variable identifiers. `identifer` and `identifer(Argument_list)` occurring to the right of the equality or inequality should be either an object or a variable.

- Any rule should not contain any undeclared objects, constants or variables. The argument of a constant must match the corresponding sort declared in constant specification: a variable must be declared to denote the element of the sort, and an object must be declared to be a member of the sort.

- `Static_law` may not contain any action identifer.

- In `Dynamic_law`, the `Atom` before `if` may not contain any fluent declared as statically determined fluent or rigid fluent.

- The `Atom_list` after `impossible` does not contain any action identifers.

- The `identifer` after `inertial` may not contain action identifer.

- The `Atom_list` after `nonexecutable` must contain at least one action identifer.

Appendix 2

Source Code of The Reactive Control System

In the source code below, the following symbols are used:

- `link(node,node,valve)` expresses that there exists a directed linkage from one node to the other through a valve.
- `direction(jet,direct)` describes the direction of a jet.
- `pair_of_jets(jet,jet)` expresses that two jets are paired.
- `has_leak(valve)` expresses that a valve has a leak.
- `done(maneuver,system)` expresses that a system is ready to perform certain maneuver task.
- `controls(switch,valve,circuit)` expresses that a switch controls a valve through a circuit.
- `control(switch,valve)` expresses that a switch controls a valve.
- `bad_circuitry(valve)` expresses that a valve has a bad circuit.
- `stuck(device,state)` expresses that a device is stuck in a state.

- `stuckd(device)` expresses that a device is stuck.
- `basic_command(command,valve,state)` expresses that a basic computer command makes a valve into certain position.
- `general_command(command,valve,state)` expresses that a computer command (including basic command and compound command) makes a valve into certain position.
- `leaking(node)` expresses that a node is leaking.
- `ready_to_fire(jet)` expresses that a jet is ready to fire.
- `damaged(jet)` expresses that a jet is damaged.
- `maneuver_of(maneuver,system)` expresses that an RCS system can perform certain maneuver task.
- `in_state(device)` is an inertial multi-valued fluent with domain being the set of `state`. It expresses that a device is in certain position.
- `ab_input(valve)` is a regular boolean fluent that expresses that a valve has an abnormal input.
- `pressurized_by(node,tank)` is a regular boolean fluent that expresses that a node is pressurized by a tank.
- `flip(switch,state)` expresses that a switch is flipped into certain position.

- `issue(command)` expresses that a command is issued by the computer.

The Source Code:

```
:- sorts


node >> miscnode;

node >> tank;

node >> jet;

    jet >> leftjet;

    jet >> rightjet;

    jet >> forwardjet;

    nontank >> jet;

nontank >> miscnode;

device >> valve;

dummyvalve >> dummyleftvalve;

dummyvalve >> dummyrightvalve;

dummyvalve >> dummyforwardvalve;

valve >> forward_valve >> dummyforwardvalve;

valve >> left_valve >> dummyleftvalve;

valve >> right_valve>> dummyrightvalve;

valve >> dummyvalve;

device >> switch;

switch >> left_switch;

switch >> right_switch;

switch >> forward_switch;
```

```
circuit;
    direct;
    state;
    maneuver;
command;
    command >> left_command;
    command >> right_command;
    command >> forward_command;
    system.

:- objects
%nodes in forward RCS

    ffj,ff12j,ff345j,ff345j :: miscnode;
ffha,ffhb,ffi12,ffi345,ffm1,ffm2,ffm3,ffm4,ffm5::forward_valve;

%nodes and valves on fuel lines
%nodes in left RCS

   lfj,lf12j,lf345j,l2l,fxfeed,lf12j :: miscnode;
   lfha,lfhb,lfi12,lfi345a,lfi345b,lfm1,lfm2,lfx12,lfm3,lfm4,lfm5,
   lfx345,lox12 :: left_valve;
```

```
%nodes in right RCS

rfj,rf12j,rf345j:: miscnode;
rfha,rfhb,rfi12,rfi345a,rfi345b,rfm1,rfm2,rfx12,rfm3,rfm4,rfm5,
    rfx345 :: right_valve;

% nodes and valves on oxidizer lines

%nodes in forward RCS

 foj,fo12j,fo345j,f1u,f1f,f1l,f1d,f2u,f2f,f2r,f2d,f3u,f3f,f3l,f3d,
    f4r,f4d,f5l,f5r :: miscnode;
foha,fohb,foi12,foi345,fom1,fom2,fom3,fom4,fom5 :: forward_valve;

%nodes in left RCS

loj,lo12j,lo345j,l1u,l1a,l1l,l2u,l2d,l2l,oxfeed,l3a,l3d,l3l,l4u,
    l4l,l5d,l5l :: miscnode;
loha,lohb,loi12,loi345a,loi345b,lom1,lom2,lox2,lom3,lom4,lom5,
    lox345 :: left_valve;

%nodes in right RCS
```

```
roj,ro12j,ro345j,r1u,r1a,r1r,r2u,r2d,r2r,ro12j,r3a,r3d,r3r,r4u,
    r4d,r5d,r5r :: miscnode;
roha,rohb,roi12,roi345a,roi345b,rom1,rom2,rox12,rom3,rom4,rom5,
    rox345 :: right_valve;

% dummy valves

ffdummy, fodummy :: dummyforwardvalve;
lfdummy, lodummy :: dummyleftvalve;
rfdummy, rodummy :: dummyrightvalve;

% switches

fha,fhb,fi12,fi345,fm1,fm2,fm3,fm4,fm5:: forward_switch;
lha,lhb,li12,li345a,li345b,lm1,lm2,lm3,lm4,lm5,lx12,lx345 :: left_switch;
rha,rhb,ri12,ri345a,ri345b,rm1,rm2,rm3,rm4,rm5,rx12,rx345 :: right_switch;

fwd_rcs,left_rcs,right_rcs,ohms :: system;
ffh,ff,foh,fo,lfh,lf,loh,lo,rfh,rf,roh,ro,ohmsf,ohmso :: tank;
flu, f1f,f1l,f1d,f2u,f2f,f2r,f2d,f3u,f3f,f3l,f3d,f4r,f4d :: forwardjet;
l1u,l1a,l1l,l2u,l2l,l2d,l3a,l3l,l3d,l4u,l4l,l4d :: leftjet;

r1u,r1a,r1r,r2u,r2r,r2d,r3a,r3r,r3d,r4u,r4r,r4d :: rightjet;
```

```
forward, up, down, aft, left, right :: direct;

minus_x,plus_x,minus_y,plus_y,minus_z,plus_z,minus_roll,
    plus_roll, minus_pitch,plus_pitch,minus_yaw,plus_yaw :: maneuver;

open,closed,gpc :: state;


    fhca,fhcb,fic12,fic345,fmc1,fmc2,fmc3,fmc4,fmc5,lhca,lhcb,lic12,

    lic345a,lic345b,lmc1,lmc2,lmc3,lmc4,lmc5, lxc12, lxc345, lox12,

    rhca,rhcb,ric12,ric345a, ric345b,rmc1,rmc2,rmc3,rmc4,rmc5,rxc12,rxc345 ::

    circuit;

% control commands of RCS

% forward control commands

    open_ffha,closea_ffha,opena_ffhb,closea_ffhb,opena_fi12,opena_foi12,

    opena_ffi345,opena_foi345,open_ffm1,closea_ffm1,closeb_ffm1,open_ffm2,
    closea_ffm2,closeb_ffm2, open_ffm3,closea_ffm3, closeb_ffm3,

    open_ffm4,closea_ffm4,closeb_ffm4 :: forward_command;

% left control commands

    opena_lfha,closea_lfha,opena_lfhb,closea_lfhb,opena_li12,openb_lfi12,

    openb_loi12,opena_lx12,openb_lfx12,open_lfi345a,close_lfi345a,

    open_lfi345b,open_loi345b,close_lfi345b,close_loi345b,open_lfx345,

    open_lox345,close_lfx345,close_lox345,open_lfm1,closea_lfm1,closeb_lfm1,

    open_lfm2,closea_lfm2,closeb_lfm2,open_lfm3,closea_lfm3,closeb_lfm3,

    open_lfm4,closea_lfm4,closeb_lfm4 :: left_command;

% right control commands

    opena_rfha,closea_rfha,opena_rfhb,closea_rfhb,opena_ri12,openb_roi12,
    opena_rx12,openb_rfx12,openb_rox12,open_rfi345a,open_roi345a,

    open_rfi345b,open_roi345b,close_rfi345b,close_roi345b,open_rfx345,

    open_rox345,close_rox345,open_rfm1,closea_rmf1,closeb_rfm1,open_rfm2,

    closea_rmf2,openb_lox12,open_loi345a,close_loi345a,openb_rfi12,

    close_roi345a,close_rfi345a,closea_rfm1,closea_rfm2,closea_rfm3,

    closeb_rfm2,open_rfm3,closea_rmf3,closeb_rfm3,open_rfm4,closea_rfm4,

    closeb_rfm4,close_rfx345:: right_command;

% compound commands

    closea_fi12_closeb_ffi12,opena_ffm5_openb_ffm5,closea_ffm5_closeb_ffm5,

    closea_fi12_closeb_foi12 :: forward_command;



    closea_li12_closeb_lfi12,close_lx12_close_lfx12,opena_lfm5_openb_lfm5,
    closea_lfm5_closeb_lfm5,closea_ri12_closeb_rfi12,close_rx12_close_rfx12,

    closea_li12,_closeb_loi12,close_lx12_close_lox12,closea_li12_closeb_loi12
    :: left_command;



    closea_ri12_closeb_roi12,close_rx12_close_rox12,opena_rfm5_openb_rfm5,
    closea_rfm5_closeb_rfm5 :: right_command.

:- variables

NT :: nontank;

N, N1,N2,N3 :: node;

    J,J1,J2 :: jet;

    LJ :: leftjet;

    RJ :: rightjet;

    FJ, FJ1, FJ2 :: forwardjet;
    T, T1,T2, TK1,TK2 :: tank;

V, V1,V2 :: valve;

LV, LV1, LV2 :: left_valve;

RV, RV1, RV2 :: right_valve;

FV, FV1, FV2 :: forward_valve;

    Dv :: dummyvalve;

Swi, Swi1,Swi2 :: switch;

LSwi, LSwi1, LSwi2 :: left_switch;

RSwi, RSwi1, RSwi2 :: right_switch;

FSwi, FSwi1, FSwi2 :: forward_switch;

De :: device;
```

```
R,S, S1,S2,R1,R2 :: system;

    St1,St2 :: state;

    X, X1, X2 :: maneuver;

    D, D1,D2 :: direct;

C :: command;

LC, LC1, LC2 :: left_command;

RC, RC1, RC2 :: right_command;

FC, FC1, FC2 :: forward_command;

    Ci :: circuit.

:- constants

link(node,node,valve), direction(jet,direct),

    pair_of_jets(jet,jet), has_leak(valve), done(maneuver,system)
    :: rigidFluent;


controls(switch,valve,circuit), control(switch,valve),

    bad_circuitry(valve), stuck(device,state),stuckd(device),

basic_command(command,valve,state),

    general_command(command,valve,state):: rigidFluent;


    leaking(node),ready_to_fire(jet),damaged(jet),

    maneuver_of(maneuver,system)::sdFluent;


in_state(device):: inertialFluent(state);

    ab_input(valve) :: simpleFluent;

    pressurized_by(node,tank) :: simpleFluent;
    flip(switch,state),issue(command) :: action.

% topology of the plumbing system

default -link(N1,N2,V).

% links for fuel line, forward RCS
link(ffh,ff,ffha). link(ffh,ff,ffhb). link(ff,ffj,ffdummy).
link(ffj,ff12j,ffi12). link(ffj,ff345j,ffi345).
link(ff12j,f1u,ffm1). link(ff12j,f1f,ffm1).
link(ff12j,f1l,ffm1). link(ff12j,f1d,ffm1).
link(ff12j,f2u,ffm2). link(ff12j,f2f,ffm2).
link(ff12j,f2r,ffm2). link(ff12j,f2d,ffm2).
link(ff345j,f3u,ffm3). link(ff345j,f3f,ffm3).
link(ff345j,f3l,ffm3). link(ff345j,f3d,ffm3).
link(ff345j,f4r,ffm4). link(ff345j,f4d,ffm4).
link(ff345j,f5l,ffm5). link(ff345j,f5r,ffm5).

% Left RCS

link(lfh,lf,lfha). link(lfh,lf,lfhb). link(lf,lfj,lfdummy).
link(lfj,lf12j,lfi12). link(lfj,lf345j,lfi345a).
link(lfj,lf345j,lfi345b). link(lf12j,l1u,lfm1).
link(lf12j,l1a,lfm1). link(lf12j,l1l,lfm1).
link(lf12j,l2u,lfm2). link(lf12j,l2d,lfm2).
link(lf12j,l2l,lfm2). link(lf12j,fxfeed,lfx12).
link(fxfeed,lf12j,lfx12). link(lf345j,l3a,lfm3).
link(lf345j,l3d,lfm3). link(lf345j,l3l,lfm3).
link(lf345j,l4u,lfm4). link(lf345j,l4d,lfm4).
link(lf345j,l4l,lfm4). link(lf345j,l5d,lfm5).
link(lf345j,l5l,lfm5). link(lf345j,fxfeed,lfx345).
link(fxfeed,lf345j,lfx345).

% Right RCS

link(rfh,rf,rfha). link(rfh,rf,rfhb). link(rf,rfj,rfdummy).
link(rfj,rf12j,rfi12). link(rfj,rf345j,rfi345a).
link(rfj,rf345j,rfi345b). link(rf12j,r1u,rfm1).
link(rf12j,r1a,rfm1). link(rf12j,r1r,rfm1).
link(rf12j,r2u,rfm2). link(rf12j,r2d,rfm2).
link(rf12j,r2r,rfm2). link(rf12j,fxfeed,rfx12).
link(fxfeed,rf12j,rfx12). link(rf345j,r3a,rfm3).
link(rf345j,r3d,rfm3). link(rf345j,r3r,rfm3).
link(rf345j,r4u,rfm4). link(rf345j,r4d,rfm4).
link(rf345j,r4r,rfm4). link(rf345,r5d,rfm5).
link(rf345j,r5r,rfm5). link(rf345j,fxfeed,rfx345).
link(fxfeed,rf345j,rfx345).

% links for oxidizer lines. Forward RCS

link(foh,fo,foha). link(foh,fo,fohb). link(fo,foj,fodummy).
link(foj,fo12j,foi12). link(foj,fo345j,foi345).
link(fo12j,f1u,fom1). link(fo12j,f1f,fom1).
link(fo12j,f1l,fom1). link(fo12j,f1d,fom1).
link(fo12j,f2u,fom2). link(fo12j,f2f,fom2).
link(fo12j,f2r,fom2). link(fo12j,f2d,fom2).
link(fo345j,f3u,fom3). link(fo345j,f3f,fom3).
link(fo345j,f3l,fom3). link(fo345j,f3d,fom3).
link(fo345j,f4r,fom4). link(fo345j,f4d,fom4).
link(fo345j,f5l,fom5). link(fo345j,f5r,fom5).

% Left RCS

link(loh,lo,loha). link(loh,lo,lohb). link(lo,loj,lodummy).
link(loj,lo12j,loi12). link(loj,lo345j,loi345a).
link(loj,lo345j,loi345b). link(lo12j,l1u,lom1).
link(lo12j,l1a,lom1). link(lo12j,l1l,lom1).
link(lo12j,l2u,lom2). link(lo12j,l2d,lom2).
link(lo12j,l2l,lom2). link(lo12j,oxfeed,lox12).
link(oxfeed,lo12j,lox12). link(lo345j,l3a,lom3).
link(lo345j,l3d,lom3). link(lo345j,l3l,lom3).
link(lo345j,l4u,lom4). link(lo345j,l4d,lom4).
link(lo345j,l4l,lom4). link(lo345j,l5d,lom5).
link(lo345j,l5l,lom5). link(lo345j,oxfeed,lox345).
link(oxfeed,lo345j,lox345).

% Right RCS

link(roh,ro,roha). link(roh,ro,rohb). link(ro,roj,rodummy).
link(roj,ro12j,roi12). link(roj,ro345j,roi345a).
link(roj,ro345j,roi345b). link(ro12j,r1u,rom1).
link(ro12j,r1a,rom1). link(ro12j,r1r,rom1).
link(ro12j,r2u,rom2). link(ro12j,r2d,rom2).
link(ro12j,r2r,rom2). link(ro12j,oxfeed,rox12).
link(oxfeed,ro12j,rox12). link(ro345j,r3a,rom3).
link(ro345j,r3d,rom3). link(ro345j,r3r,rom3).
link(ro345j,r4u,rom4). link(ro345j,r4d,rom4).
link(ro345j,r4r,rom4). link(ro345j,r5d,rom5).
link(ro345j,r5r,rom5). link(ro345j,oxfeed,rox345).
link(oxfeed,ro345j,rox345).
```

```
% direction(J,D) iff jet J fires in direction D.

default -direction(J,D).
direction(f1f,forward). direction(f2f,forward).
direction(f3f,forward). direction(f1u,up).
direction(f2u,up). direction(f3u,up).
direction(l1u,up). direction(l2u,up).
direction(l4u,up). direction(r1u,up).
direction(r2u,up). direction(r4u,up).
direction(f1d,down). direction(f2d,down).
direction(f3d,down). direction(f4d,down).
direction(l2d,down). direction(l3d,down).
direction(l4d,down). direction(r2d,down).
direction(r3d,down). direction(r4d,down).
direction(l1a,aft). direction(l3a,aft).
direction(r1a,aft). direction(r3a,aft).
direction(f1l,left). direction(f3l,left).
direction(l1l,left). direction(l2l,left).
direction(l3l,left). direction(l4l,left).
direction(f2r,right). direction(f4r,right).
direction(r1r,right). direction(r2r,right).
direction(r3r,right). direction(r4r,right).
```

```
% The downward firing jets on the forward rcs must be used
%in pairs, one on each side.

default -pair_of_jets(J1,J2).
pair_of_jets(f1d,f2d). pair_of_jets(f1d,f4d).
pair_of_jets(f3d,f2d). pair_of_jets(f3d,f4d).

% By default a valve does not have leak and
% a node is not leaking. A node is leaking if (1)
% it is connected to another node via a valve that
% has a leak and the valve is open, or (2) it is
% connected to a leaking node and the connecting valve is open.

default -has_leak(V).
default -leaking(N).
leaking(N1) if  link(N1,N2,V), has_leak(V), in_state(V)=open.
leaking(N1) if  link(N1,N2,V), leaking(N2), in_state(V)=open.

% By default a node is not pressurized by a tank.
% A Tank is always pressurized by itself if it is once
% pressurized by itself. A non-tank node NT is pressurized
% by a tank T if it is not leaking, another node N2 is connected
% to NT via valve V, the valve is open, and N2 is pressurized by T.
% A tank node T1 is pressurized by T if another node N1 is linked to
% T1 via an open valve, and N1 is pressurized by T.

pressurized_by(T1,T1) after pressurized_by(T1,T1).
default -pressurized_by(N1,T1).
pressurized_by(NT,T) if link(N2,NT,V), in_state(V)=open,
          pressurized_by(N2,T), -leaking(NT).
pressurized_by(T1,T) if link(N2,T1,V),
                   in_state(V)=open,pressurized_by(N2,T).

% By default a jet is not damaged and not ready to fire.
% It is ready to fire if it is not damaged and pressurized
% by two different tanks.

default -damaged(J).
default -ready_to_fire(J).
ready_to_fire(J) pressurized_by(J,TK1), pressurized_by(J,TK2),
    -damaged(J), TK1\=TK2.

% By default a maneuver is not ready to perform unless
% it is known to be done. A RCS system is ready to perform
% a maneuver if one of its jet in corresponding direction is
% ready to fire. This involves the following rules which are
% self-explained:


default -done(X,S).

default -maneuver_of(X,S).

maneuver_of(X,S) if done(X,S).

maneuver_of(plus_x,left_rcs) if direction(LJ,aft),

                                ready_to_fire(LJ).

maneuver_of(plus_x,right_rcs) if direction(RJ,aft),

                                 ready_to_fire(RJ).

done(plus_x,fwd_rcs).

maneuver_of(minus_x,fwd_rcs) if   direction(FJ1,forward),

                                  direction(FJ2,forward),

                                  ready_to_fire(FJ1),

                                  ready_to_fire(FJ2), FJ1\=FJ2.

done(minus_x,left_rcs).

done(minus_x,right_rcs).

maneuver_of(plus_y,left_rcs) if direction(LJ,left),

                                ready_to_fire(LJ).

maneuver_of(plus_y,fwd_rcs) if direction(FJ,left),

                                ready_to_fire(FJ).

done(plus_y,right_rcs).

maneuver_of(minus_y,right_rcs) if direction(RJ,right),

                                ready_to_fire(RJ).
```

```
maneuver_of(minus_y,fwd_rcs) if direction(FJ,right),
                                ready_to_fire(FJ).
done(minus_y,left_rcs).
maneuver_of(plus_z,left_rcs) if direction(LJ,up),
                                ready_to_fire(LJ).
maneuver_of(plus_z,right_rcs) if direction(RJ,up),
                                ready_to_fire(RJ).
maneuver_of(plus_z,fwd_rcs) if direction(FJ,up),
                                ready_to_fire(FJ).
maneuver_of(minus_z,left_rcs) if direction(LJ,down),
                                 ready_to_fire(LJ).
maneuver_of(minus_z,right_rcs) if  direction(RJ,down),
                                   ready_to_fire(RJ).
maneuver_of(minus_z,fwd_rcs) if  pair_of_jets(FJ1,FJ2),
                                 ready_to_fire(FJ1),
                                 ready_to_fire(FJ2).
maneuver_of(plus_roll,left_rcs) if direction(LJ,down),
                                   ready_to_fire(LJ).
maneuver_of(plus_roll,right_rcs) if direction(RJ,up),
                                    ready_to_fire(RJ).
done(plus_roll,fwd_rcs).
maneuver_of(minus_roll,left_rcs) if direction(LJ,up),
                                    ready_to_fire(LJ).
```

```
maneuver_of(minus_roll,right_rcs) if direction(RJ,down),
                                     ready_to_fire(RJ).
done(minus_roll,fwd_rcs).
maneuver_of(plus_pitch,fwd_rcs) if   pair_of_jets(FJ1,FJ2),
                                     ready_to_fire(FJ1),
                                     ready_to_fire(FJ2).
maneuver_of(plus_pitch,left_rcs) if  direction(LJ,up),
                                     ready_to_fire(LJ).
maneuver_of(plus_pitch,right_rcs) if  direction(RJ,up),
                                      ready_to_fire(RJ).
maneuver_of(minus_pitch,fwd_rcs) if  direction(FJ,up),
                                     ready_to_fire(FJ).
maneuver_of(minus_pitch,left_rcs) if direction(LJ,down),
                                     ready_to_fire(LJ).
maneuver_of(minus_pitch,right_rcs) if direction(RJ,down),
                                      ready_to_fire(RJ).
maneuver_of(plus_yaw,right_rcs) if direction(RJ,right),
                                   ready_to_fire(LJ).
maneuver_of(plus_yaw,fwd_rcs) if direction(FJ,left),
                                 ready_to_fire(FJ).
done(plus_yaw,left_rcs).
maneuver_of(minus_yaw,left_rcs) if direction(LJ,left),
                                   ready_to_fire(LJ).
```

```
maneuver_of(minus_yaw,fwd_rcs) if direction(FJ,right),
                                  ready_to_fire(FJ).
done(minus_yaw,right_rcs).

control(Swi,V) if controls(Swi,V,Ci).
default -control(Swi,V).
default -controls(Swi,V,Ci).

% Forward RCS

controls(fha,ffha,fhca).controls(fha,foha,fhca).
controls(fhb,ffhb,fhcb).controls(fhb,fohb,fhcb).
controls(fi12,ffi12,fic12).controls(fi12,foi12,fic12).
controls(fi345,ffi345,fic345).controls(fi345,foi345,fic345).
controls(fm1,ffm1,fmc1).controls(fm1,fom1,fmc1).
controls(fm2,ffm2,fmc2).controls(fm2,fom2,fmc2).
controls(fm3,ffm3,fmc3).controls(fm3,fom3,fmc3).
controls(fm4,ffm4,fmc4).controls(fm4,fom4,fmc4).
controls(fm5,ffm5,fmc5).controls(fm5,fom5,fmc5).

% Left RCS

controls(lha,lfha,lhca).controls(lha,loha,lhca).
controls(lhb,lfhb,lhcb).controls(lhb,lohb,lhcb).

controls(li12,lfi12,lic12).controls(li12,loi12,lic12).

controls(li345a,lfi345a,lic345a).controls(li345a,loi345a,lic345a).

controls(li345b,lfi345b,lic345b).controls(li345b,loi345b,lic345b).

controls(lm1,lfm1,lmc1).controls(lm1,lom1,lmc1).

controls(lm2,lfm2,lmc2).controls(lm2,lom2,lmc2).

controls(lm3,lfm3,lmc3).controls(lm3,lom3,lmc3).

controls(lm4,lfm4,lmc4).controls(lm4,lom4,lmc4).

controls(lm5,lfm5,lmc5).controls(lm5,lom5,lmc5).

controls(lx12,lfx12,lxc12).controls(lx12,lox12,lxc12).

controls(lx345,lfx345,lxc345).controls(lx345,lox345,lxc345).


% Right RCS


controls(rha,rfha,rhca).controls(rha,roha,rhca).

controls(rhb,rfhb,rhcb).controls(rhb,rohb,rhcb).

controls(ri12,rfi12,ric12).controls(ri12,roi12,ric12).

controls(ri345a,rfi345a,ric345a).controls(ri345a,roi345a,ric345a).

controls(ri345b,rfi345b,ric345b).controls(ri345b,roi345b,ric345b).

controls(rm1,rfm1,rmc1).controls(rm1,rom1,rmc1).

controls(rm2,rfm2,rmc2).controls(rm2,rom2,rmc2).

controls(rm3,rfm3,rmc3).controls(rm3,rom3,rmc3).

controls(rm4,rfm4,rmc4).controls(rm4,rom4,rmc4).
controls(rm5,rfm5,rmc5).controls(rm5,rom5,rmc5).
controls(rx12,rfx12,rxc12).controls(rx12,rox12,rxc12).
controls(rx345,rfx345,rxc345).controls(rx345,rox345,rxc345).

% basic commands

default -basic_command(C,V,St1).
basic_command(opena_lfha,lfha,open).
basic_command(opena_lfha,loha,open).
basic_command(closea_lfha,lfha,closed).
basic_command(closea_lfha,loha,closed).
basic_command(opena_lfhb,lfhb,open).
basic_command(opena_lfhb,lohb,open).
basic_command(closea_lfhb,lfhb,closed).
basic_command(closea_lfhb,lohb,closed).

% Commands to control valves lfi12, loi12

basic_command(opena_li12,lfi12,open).
basic_command(opena_li12,loi12,open).
basic_command(openb_lfi12,lfi12,open).
basic_command(openb_loi12,loi12,open).
```

```
% Commands to control valves (crossfeeds) lfx12, lox12

basic_command(opena_lx12,lfx12,open).
basic_command(opena_lx12,lox12,open).
basic_command(openb_lfx12,lfx12,open).
basic_command(openb_lox12,lox12,open).

% Commands to control valves lfi345a, lfi345b, loi345a, lfi345b

basic_command(open_lfi345a,lfi345a,open).
basic_command(open_loi345a,loi345a,open).
basic_command(close_lfi345a,lfi345a,closed).
basic_command(close_loi345a,loi345a,closed).
basic_command(open_lfi345b,lfi345b,open).
basic_command(open_loi345b,loi345b,open).
basic_command(close_lfi345b,lfi345b,closed).
basic_command(close_loi345b,loi345b,closed).

% Commands to control valves (crossfeeds) lfx345, lox345

basic_command(open_lfx345,lfx345,open).
basic_command(open_lox345,lox345,open).
basic_command(close_lfx345,lfx345,closed).
basic_command(close_lox345,lox345,closed).

% Commands to control valves (manifolds) lfm1, lom1

basic_command(open_lfm1,lfm1,open).
basic_command(open_lfm1,lom1,open).
basic_command(closea_lfm1,lfm1,closed).
basic_command(closea_lfm1,lom1,closed).
basic_command(closeb_lfm1,lfm1,closed).
basic_command(closeb_lfm1,lom1,closed).

% Commands to control valves (manifolds) lfm2, lom2

basic_command(open_lfm2,lfm2,open).
basic_command(open_lfm2,lom2,open).
basic_command(closea_lfm2,lfm2,closed).
basic_command(closea_lfm2,lom2,closed).
basic_command(closeb_lfm2,lfm2,closed).
basic_command(closeb_lfm2,lom2,closed).

% Commands to control valves (manifolds) lfm3, lom3

basic_command(open_lfm3,lfm3,open).
basic_command(open_lfm3,lom3,open).

basic_command(closea_lfm3,lfm3,closed).

basic_command(closea_lfm3,lom3,closed).

basic_command(closeb_lfm3,lfm3,closed).

basic_command(closeb_lfm3,lom3,closed).


% Commands to control valves (manifolds) lfm4, lom4


basic_command(open_lfm4,lfm4,open).

basic_command(open_lfm4,lom4,open).

basic_command(closea_lfm4,lfm4,closed).

basic_command(closea_lfm4,lom4,closed).

basic_command(closeb_lfm4,lfm4,closed).

basic_command(closeb_lfm4,lom4,closed).


% Right RCS


% Commands to control valves rfha, rohb, rfha, rohb


basic_command(opena_rfha,rfha,open).

basic_command(opena_rfha,roha,open).

basic_command(closea_rfha,rfha,closed).

basic_command(closea_rfha,roha,closed).
basic_command(opena_rfhb,rfhb,open).

basic_command(opena_rfhb,rohb,open).

basic_command(closea_rfhb,rfhb,closed).

basic_command(closea_rfhb,rohb,closed).


% Commands to control valves rfi12, roi12


basic_command(opena_ri12,rfi12,open).

basic_command(opena_ri12,roi12,open).

basic_command(openb_rfi12,rfi12,open).

basic_command(openb_roi12,roi12,open).


% Commands to control valves (crossfeeds) rfx12, rox12


basic_command(opena_rx12,rfx12,open).

basic_command(opena_rx12,rox12,open).

basic_command(openb_rfx12,rfx12,open).

basic_command(openb_rox12,rox12,open).


% Commands to control valves rfi345a, rfi345b, roi345a, fi345b


basic_command(open_rfi345a,rfi345a,open).

basic_command(open_roi345a,roi345a,open).
basic_command(close_rfi345a,rfi345a,closed).

basic_command(close_roi345a,roi345a,closed).

basic_command(open_rfi345b,rfi345b,open).

basic_command(open_roi345b,roi345b,open).

basic_command(close_rfi345b,rfi345b,closed).

basic_command(close_roi345b,roi345b,closed).


% Commands to control valves (crossfeeds) rfx345, rox345


basic_command(open_rfx345,rfx345,open).

basic_command(open_rox345,rox345,open).

basic_command(close_rfx345,rfx345,closed).

basic_command(close_rox345,rox345,closed).


% Commands to control valves (manifolds) rfm1, rom1


basic_command(open_rfm1,rfm1,open).

basic_command(open_rfm1,rom1,open).

basic_command(closea_rfm1,rfm1,closed).

basic_command(closea_rfm1,rom1,closed).

basic_command(closeb_rfm1,rfm1,closed).

basic_command(closeb_rfm1,rom1,closed).
```

```
% Commands to control valves (manifolds) rfm2, rom2

basic_command(open_rfm2,rfm2,open).
basic_command(open_rfm2,rom2,open).
basic_command(closea_rfm2,rfm2,closed).
basic_command(closea_rfm2,rom2,closed).
basic_command(closeb_rfm2,rfm2,closed).
basic_command(closeb_rfm2,rom2,closed).

% Commands to control valves (manifolds) rfm3, rom3

basic_command(open_rfm3,rfm3,open).
basic_command(open_rfm3,rom3,open).
basic_command(closea_rfm3,rfm3,closed).
basic_command(closea_rfm3,rom3,closed).
basic_command(closeb_rfm3,rfm3,closed).
basic_command(closeb_rfm3,rom3,closed).

% Commands to control valves (manifolds) rfm4, rom4

basic_command(open_rfm4,rfm4,open).
basic_command(open_rfm4,rom4,open).
basic_command(closea_rfm4,rfm4,closed).
basic_command(closea_rfm4,rom4,closed).

basic_command(closeb_rfm4,rfm4,closed).

basic_command(closeb_rfm4,rom4,closed).


% general command


default -general_command(C,V,St1).

general_command(C,V,St1) if basic_command(C,V,St1).

general_command(closea_fi12_closeb_ffi12,ffi12,closed).

general_command(closea_fi12_closeb_foi12,foi12,closed).

general_command(opena_ffm5_openb_ffm5,ffm5,open).

general_command(opena_ffm5_openb_ffm5,fom5,open).

general_command(closea_ffm5_closeb_ffm5,ffm5,closed).

general_command(closea_ffm5_closeb_ffm5,fom5,closed).

general_command(closea_li12_closeb_lfi12,lfi12,closed).

general_command(closea_li12_closeb_loi12,loi12,closed).

general_command(close_lx12_close_lfx12,lfx12,closed).

general_command(close_lx12_close_lox12,lox12,closed).

general_command(opena_lfm5_openb_lfm5,lfm5,open).

general_command(opena_lfm5_openb_lfm5,lom5,open).

general_command(closea_lfm5_closeb_lfm5,lfm5,closed).

general_command(closea_lfm5_closeb_lfm5,lom5,closed).

general_command(closea_ri12_closeb_rfi12,rfi12,closed).
general_command(closea_ri12_closeb_roi12,roi12,closed).
general_command(close_rx12_close_rfx12,rfx12,closed).
general_command(close_rx12_close_rox12,rox12,closed).
general_command(opena_rfm5_openb_rfm5,rfm5,open).
general_command(opena_rfm5_openb_rfm5,rom5,open).
general_command(closea_rfm5_closeb_rfm5,rfm5,closed).
general_command(closea_rfm5_closeb_rfm5,rom5,closed).

% If a device is stuck at certain position, it is stuck;
% by default a device is not stuck. All dummy valves are stuck
% at open position. By default a valve doesn’t have a bad circuitry,
% and is not an abnormal input.

stuckd(De) if stuck(De,St1).
stuck(Dv,open).
default -stuck(De,St1).
default -stuckd(De).
default -bad_circuitry(V).
default -ab_input(V).



% A valve is in a state if the state is not {\tt gpc} state,
% the switch that controls valve is in the state and the valve
% is not stuck, doesn't have bad circuitry, or is an abnormal input:
in_state(V)=St1 if  control(Swi,V),
                    -ab_input(V),
                    -stuckd(V),
                    -bad_circuitry(V)
            St1\=gpc.

% Flipping a unstuck switch to a state causes the switch to be
% in this state. It is not executable if the switch has already
% been in this state.

flip(Swi,St1) causes in_state(Swi)=St1 if  -stuckd(Swi).
nonexecutable flip(Swi,St1) if in_state(Swi)=St1.



% Issuing a command C causes a valve V to be in a state St1 if
% the valve is controlled by a switch that is in gpc state, and C
% is a general command that changes V into state St1, V is not
% stuck or has a bad circuitry.

issue(C) causes in_state(V)=St1 if control(Swi,V),
                in_state(Swi)=gpc,
                general_command(C,V,St1),
                -stuckd(V),
                 -bad_circuitry(V).

% If switch Swi controlling a valve V is in some state
% St1 (open or closed) and all computer command required to
% move V to some state St2 different from St1 were issued,
% then the input to V is considered abnormal, i.e. the state of V
% is undefined in high-level valve control system.

issue(C) causes ab_input(V) if control(Swi,V),
                               in_state(Swi)=St1,
                               St1\=gpc,
                               general_command(C,V,St2),
                               St2\=St1
                               -bad_circuitry(V).

% Finally, at any time, for each RCS system, only one action can be performed:

nonexecutable flip(LSwi1,St1), flip(LSwi2,St2) where LSwi1\=LSwi2.
nonexecutable flip(RSwi1,St1), flip(RSwi2,St2) where RSwi1\=RSwi2.
nonexecutable flip(FSwi1,St1), flip(FSwi2,St2) where FSwi1\=FSwi2.
nonexecutable issue(LC1), issue(LC2) where LC1\=LC2.
nonexecutable issue(RC1), issue(RC2) where RC1\=RC2.
nonexecutable issue(FC1), issue(FC2) where FC1\=FC2.

nonexecutable flip(LSwi,St1), issue(LC).

nonexecutable flip(RSwi,St1), issue(RC).

nonexecutable flip(FSwi,St1), issue(FC).
```

Appendix 3

Source Code for the Robot Task Planning Domain

3.1 Example from Section 12.2

```
:- sorts

room>>office;

door;

person.


:- objects

alice,bob,carol,dan :: person;

o1,o2,o3  :: office;

cor   :: room;

d1,d2,d3 :: door.


:- variables

D,D1,D2,D3  :: door;

P,P1,P2,P3 :: person;

R,R1,R2,R3  :: room;

O,O1,O2,O3 :: office.
```

```
:- constants
hasdoor(room,door), acc(room,door,room),
    knows(person,person), passto(person,person),
    relatedto(person,person) :: sdFluent;

inside(person,room),beside(door),facing(door),
knowinside(person,room),served(person):: simpleFluent;

    loc :: simpleFluent(room);

open(door), visiting(person) :: simpleFluent;

approach(door), gothrough(door), callforopen(door),
goto(person), askploc(person,person),serve(room) :: action.

inertial inside(P1,R1).
inertial beside(D).
inertial facing(D).
inertial knowinside(P,R).
inertial served(P).
inertial loc.
```

```
default -hasdoor(R,D).

caused hasdoor(o1,d1).

caused hasdoor(o2,d2).

caused hasdoor(o3,d3).

caused hasdoor(cor,d1).

caused hasdoor(cor,d2).

caused hasdoor(cor,d3).


default -open(D).

default -acc(R1,D,R2).


caused acc(R,D,cor) if hasdoor(R,D).

caused acc(R1,D,R2) if acc(R2,D,R1).


default -relatedto(P1,P2).

default -passto(P1,P2).



caused relatedto(P1,P2) if passto(P2,P1).

caused relatedto(P3,P1) if passto(P1,P2), relatedto(P3,P2).


caused knows(alice,dan).

default -knows(P1,P2).
```

```
caused -beside(D1) if beside(D2) where D2\=D1.
caused -facing(D2) if facing(D1) where D2\=D1.
caused beside(D) if facing(D).

caused -inside(P,R1) if inside(P,R2) where R2\=R1.
caused -knowinside(P,R2) if knowinside(P,R1) where R2\=R1.

caused inside(P,R) if knowinside(P,R).

default -open(D).
default -visiting(P).

approach(D) causes facing(D).
nonexecutable approach(D) if loc=R, -hasdoor(R,D).
nonexecutable approach(D) if facing(D).

gothrough(D) causes loc=R if acc(R1,D,R), loc=R1 where R\=R1.
gothrough(D) causes -facing(D).
nonexecutable gothrough(D) if -facing(D).
nonexecutable gothrough(D) if -open(D).
nonexecutable gothrough(D) if loc=R, -hasdoor(R,D).
```

```
callforopen(D) causes open(D).
nonexecutable callforopen(D) if -facing(D).
nonexecutable callforopen(D) if open(D).

goto(P) causes visiting(P).
goto(P) causes served(P).
caused served(P1) if served(P), relatedto(P,P1).
nonexecutable goto(P) if loc=R, -knowinside(P,R).
nonexecutable goto(P) if loc\=R, knowinside(P,R).

askploc(P1,P) causes knowinside(P,R) if inside(P,R).
nonexecutable askploc(P1,P) if -visiting(P1).
nonexecutable askploc(P1,P) if visiting(P1), -knows(P1,P).
nonexecutable askploc(P1,P) if knowinside(P,R).

noconcurrency.
```

3.2 A Robot in the GDC Building

The source code shown in this section is used in a mobile robot that navigates through GDC 3NE wing.

```
:- sorts
```

```
room>>office;
door;
person.

:- objects
peterstone,danaballard,raymooney,stacymiller :: person;


    o3_508,o3_510,o3_512,o3_416,l3_414a,s3_516,l3_436,o3_428,
    o3_426,l3_414b,l3_414,o3_402,o3_412,o3_502,o3_404,o3_418,
    o3_420,o3_422,o3_430,o3_432  :: office;


cor :: room;
    d3_508,d3_510,d3_512,d3_416,d3_414a1,d3_414a2,d3_414a3,
    d3_5161,d3_5162,d3_4361,d3_4362,d3_428,d3_426,d3_414b1,
    d3_414b2,d3_414b3,d3_402,d3_412,d3_502,d3_404,d3_418,
    d3_420,d3_422,d3_430,d3_432 :: door.

:- variables
D,D1,D2,D3  :: door;
P,P1,P2,P3 :: person;
R,R1,R2,R3   :: room;
O,O1,O2,O3 :: office.
```

```
:- constants

hasdoor(room,door), acc(room,door,room),

    knows(person,person), passto(person,person),

    relatedto(person,person)                :: rigidFluent;


inside(person,room),beside(door),facing(door),

knowinside(person,room),served(person):: inertialFluent;


    loc :: inertialFluent(room);

open(door), visiting(person) :: simpleFluent;

approach(door), gothrough(door), callforopen(door),

goto(person), askploc(person,person),serve(room) :: exogenousAction.


default -hasdoor(R,D).

hasdoor(o3_508,d3_508).

hasdoor(o3_510,d3_510).

hasdoor(o3_512,d3_512).

hasdoor(o3_416,d3_416).

hasdoor(l3_414a,d3_414a1).

hasdoor(l3_414a,d3_414a2).

hasdoor(l3_414a,d3_414a3).

hasdoor(s3_516,d3_5161).

hasdoor(s3_516,d3_5162).
hasdoor(l3_436,d3_4361).

hasdoor(l3_436,d3_4362).

hasdoor(o3_426,d3_426).

hasdoor(o3_428,d3_428).

hasdoor(l3_414b,d3_414b1).

hasdoor(l3_414b,d3_414b2).

hasdoor(l3_414b,d3_414b3).

hasdoor(o3_402,d3_402).

hasdoor(o3_412,d3_412).

hasdoor(o3_502,d3_502).

hasdoor(o3_404,d3_404).

hasdoor(o3_418,d3_418).

hasdoor(o3_420,d3_420).

hasdoor(o3_422,d3_422).

hasdoor(o3_430,d3_430).

hasdoor(o3_432,d3_432).

hasdoor(cor,D) where D\=d3_414a3, D\=d3_414b3.


default -open(D).

default -acc(R1,D,R2).


acc(R,D,cor) if hasdoor(R,D) where D\=d3_414a3, D\=d3_414b3.

acc(R1,D,R2) if acc(R2,D,R1).
acc(l3_414a,d3_414a3,l3_414).

acc(l3_414b,d3_414b3,l3_414).


default -relatedto(P1,P2).

default -passto(P1,P2).


relatedto(P1,P2) if passto(P2,P1).

relatedto(P3,P1) if passto(P1,P2), relatedto(P3,P2).


knows(peterstone,danaballard).

knows(stacymiller,raymooney).

default -knows(P1,P2).


caused -beside(D1) if beside(D2) where D2\=D1.

caused -facing(D2) if facing(D1) where D2\=D1.

caused beside(D) if facing(D).


caused -inside(P,R1) if inside(P,R2) where R2\=R1.

caused -knowinside(P,R2) if knowinside(P,R1) where R2\=R1.


caused inside(P,R) if knowinside(P,R).


default -open(D).
default -visiting(P).

approach(D) causes facing(D).
nonexecutable approach(D) if loc=R, -hasdoor(R,D).
nonexecutable approach(D) if facing(D).

gothrough(D) causes loc=R if acc(R1,D,R), loc=R1 where R\=R1.
gothrough(D) causes -facing(D).
nonexecutable gothrough(D) if -facing(D).
nonexecutable gothrough(D) if -open(D).
nonexecutable gothrough(D) if loc=R, -hasdoor(R,D).

callforopen(D) causes open(D).
nonexecutable callforopen(D) if -facing(D).
nonexecutable callforopen(D) if open(D).

goto(P) causes visiting(P).
goto(P) causes served(P).
caused served(P1) if served(P), relatedto(P,P1).
nonexecutable goto(P) if loc=R, -knowinside(P,R).
nonexecutable goto(P) if loc\=R, knowinside(P,R).

askploc(P1,P) causes knowinside(P,R) if inside(P,R).
```

```
nonexecutable askploc(P1,P) if -visiting(P1).

nonexecutable askploc(P1,P) if visiting(P1), -knows(P1,P).

nonexecutable askploc(P1,P) if knowinside(P,R).


noconcurrency.
```

Bibliography

[1] 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.

[2] Joseph Babb and Joohyung Lee. Cplus2asp: Computing action language C+ in answer set programming. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2013.

[3] Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. Towards answer set programming with sorts. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2013.

[4] Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with a-prolog. *Theory and Practice of Logic Programming*, 3(4-5):425–461, 2003.

[5] Michael Bartholomew and Joohyung Lee. Stable models of formulas with intensional functions. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2012.

[6] Nicole Bidoit and Christine Froidevaux. Minimalism subsumes default logic and circumscription in stratified logic programming. In *Proceedings LICS-87*, pages 89–97, 1987.

[7] Ozan Caldiran, Kadir Haspalamutgil, Abdullah Ok, Can Palaz, Esra Erdem, and Volkan Patoglu. Bridging the gap between high-level reasoning and low-level control. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2009.

[8] Michael Casolary and Joohyung Lee. Representing the language of the causal calculator in answer set programming. In Gallagher and Gelfond [23], pages 51–61.

[9] Xiaoping Chen, Jianmin Ji, Jiehui Jiang, Guoqiang Jin, Feng Wang, and Jiongkun Xie. Developing high-level cognitive functions for service robots. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2010.

[10] Sandeep Chintabathina, Michael Gelfond, and Richard Watson. Modeling hybrid domains using process description language[1]. In *Proceedings of Workshop on Answer Set Programming: Advances in Theory and Implementation (ASP'05)*, 2005.

[11] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York,

[1]`http://ceur-ws.org/vol-142/page303.pdf`

1978.

[12] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer set planning under action costs. *J. Artif. Intell. Res.(JAIR)*, 19:25–71, 2003.

[13] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[14] Selim T. Erdoğan and Vladimir Lifschitz. Actions as special cases. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 377–387, 2006.

[15] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[16] Paolo Ferraris. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 119–131, 2005.

[17] Paolo Ferraris. A logic program characterization of causal theories. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 366–371, 2007.

[18] Paolo Ferraris, Joohyung Lee, Yuliya Lierler, Paolo Lifschitz, and Fangkai Yang. Representing first-order causal theories by logic programs. *Theory and Practice of Logic Programming*, 2010. To appear.

[19] Paolo Ferraris, Joohyung Lee, Yuliya Lierler, Vladimir Lifschitz, and Fangkai Yang. Representing first-order causal theories by logic programs. *Theory and Practice of Logic Programming*, 12(3):383–412, 2012.

[20] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.

[21] Paolo Ferraris, Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Symmetric splitting in the general theory of stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 797–803, 2009.

[22] 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.

[23] John P. Gallagher and Michael Gelfond, editors. *Technical Communications of the 27th International Conference on Logic Programming, ICLP 2011, July 6-10, 2011, Lexington, Kentucky, USA*, volume 11 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[24] Martin Gebser, Torsten Grote, and Torsten Schaub. Coala: a compiler from action languages to ASP. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, 2010.

[25] Hector Geffner. Causal theories for nonmonotonic reasoning. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages

524–530. AAAI Press, 1990.

[26] Michael Gelfond and Daniela Inclezan. Yet another modular action language. In *Proceedings of the Second International Workshop on Software Engineering for Answer Set Programming*[2], pages 64–78, 2009.

[27] 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. MIT Press, 1988.

[28] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[29] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[30] Michael Gelfond and Vladimir Lifschitz. Action languages[3]. *Electronic Transactions on Artificial Intelligence*, 3:195–210, 1998.

[31] Michael Gelfond and Vladimir Lifschitz. The common core of action languages $\mathcal{B}$ and $\mathcal{C}$[4]. In *Working Notes of the International Workshop on Nonmonotonic Reasoning (NMR)*. 2012.

[2]`http://www.sea09.cs.bath.ac.uk/downloads/sea09proceedings.pdf`
[3]`http://www.ep.liu.se/ea/cis/1998/016/`
[4]`http://www.cs.utexas.edu/users/vl/papers/bc.pdf`

[32] Michael Gelfond, Vladimir Lifschitz, Halina Przymusińska, and Mirosław Truszczyński. Disjunctive defaults. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 230–237, 1991.

[33] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

[34] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 623–630. AAAI Press, 1998.

[35] Vladimir Glivenko. Sur quelques points de la logique de M. Brouwer. *Académie Royale de Belgique. Bulletins de la Classe des Sciences, se'rie 5*, 15:183–188, 1929.

[36] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[37] Piyush Khandelwal, Fangkai Yang, Matteo Leonetti, Vladimir Lifschitz, and Peter Stone. Planning in action language BC while learning action costs for mobile robots. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*, June 2014. to appear.

[38] Joohyung Lee. *Automated Reasoning about Actions*[5]. PhD thesis, University of Texas at Austin, 2005.

[39] Joohyung Lee, Yuliya Lierler, Vladimir Lifschitz, and Fangkai Yang. Representing synonymity in causal logic and in logic programming[6]. In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)*, 2010.

[40] Joohyung Lee, Vladimir Lifschitz, and Fangkai Yang. Action language $\mathcal{BC}$: A preliminary report. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.

[41] Joohyung Lee and Yunsong Meng. First-order stable model semantics and first-order loop formulas. *J. Artif. Intell. Res. (JAIR)*, 42:125–180, 2011.

[42] Joohyung Lee and Ravi Palla. System F2LP — computing answer sets of first-order formulas. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2009.

[43] Vladimir Lifschitz. Circumscription. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1994.

[5] `http://peace.eas.asu.edu/joolee/papers/dissertation.pdf`
[6] `http://userweb.cs.utexas.edu/users/vl/papers/syn.pdf`

[44] Vladimir Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.

[45] Vladimir Lifschitz. Logic programs with intensional functions. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2012.

[46] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[47] Vladimir Lifschitz and Wanwan Ren. A modular action description language. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 853–859, 2006.

[48] Vladimir Lifschitz and Wanwan Ren. The semantics of variables in action descriptions. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 2007.

[49] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[50] Vladimir Lifschitz and Fangkai Yang. Translating first-order causal theories into answer set programming. In *Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA)*, 2010.

[51] Vladimir Lifschitz and Fangkai Yang. Eliminating function symbols from a nonmonotonic causal theory. In *Knowing, Reasoning, and Acting: Essays in Honour of Hector J. Levesque*. College Publications, 2011.

[52] Vladimir Lifschitz and Fangkai Yang. Lloyd-topor completion and general stable models. In *Proceedings of 5th International Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'2012)*, 2012.

[53] Vladimir Lifschitz and Fangkai Yang. Functional completion. *Journal of Applied Nonmonotonic Logic: Special Issue in Honor of David Pearce*, 23(1-2), 2013.

[54] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1985–1991, 1995.

[55] Fangzhen Lin and Yisong Wang. Answer set programming with functions. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2008.

[56] John Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[57] John Lloyd and Rodney Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[58] Norman McCain. *Causality in Commonsense Reasoning about Actions*[7]. PhD thesis, University of Texas at Austin, 1997.

[59] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 460–465, 1997.

[60] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39,171–172, 1980.

[61] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.

[62] 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.

[63] Grigori Mints. *A Short Introduction to Intuitionistic Logic*. Kluwer, 2000.

[64] Robert Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[65] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space

[7]`ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.gz`

Shuttle. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 169–183, 2001.

[66] Edwin Pednault. ADL and the state-transition model of action. *Journal of Logic and Computation*, 4:467–512, 1994.

[67] 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.

[68] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[69] 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.

[70] Wanwan Ren. *A Modular Language for Describing Actions*[8]. PhD thesis, University of Texas at Austin, 2009.

[71] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[8]`http://www.cs.utexas.edu/users/rww6/dissertation.pdf`

[72] Miroslaw Truszczynski. Connecting first-order asp and the logic fo(id) through reducts. 2012.

[73] Fangkai Yang, Piyush Khandelwal, Matteo Leonetti, and Peter Stone. Planning in answer set programming while learning action costs for mobile robots. In *AAAI Spring 2014 Symposium on Knowledge Representation and Reasoning in Robotics (AAAI-SSS)*, March 2014.

Vita

Fangkai Yang was born in Hefei, Anhui Province, China in 1983, and lived in Hefei until moving to Austin in 2008. He attended No.1 Senior High School of Hefei from 1998 to 2001, Hefei Univeristy of Technology from 2001 to 2005 where he obtained the degree of Bachelor of Engineering from the Department of Computer Science and Enginnering. After that, he joined the Multi-agent System Lab in Department of Computer Science, University of Science and Technology of China, under the supervision of Professor Xiaoping Chen. From there he obtained his Master of Science degree in 2008. From September of 2008 he was enrolled in the doctoral program in Department of Computer Science, the University of Texas at Austin. From May 2014, he starts working for Schlumberger Technology Corporation, in Houston, Texas.

Permanent email address: `wolfgang.yang@gmail.com`

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.