University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Execution Mechanisms for the Action Language $\mathcal{C}+$

Robert Anthony Craven

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, September 2006

Abstract

The action language $\mathcal{C}+$ of Giunchiglia, Lee, Lifschitz, McCain and Turner, is the most recent and expressive member of a family of knowledge representation formalisms for reasoning about action and change over time. However, frequently the reasoning proceeds very inefficiently, for large domains collapsing altogether; most often the reason for this is that in determining the truth of a fluent at a given time, one must determine the truth of all fluents at all times: the most irrelevant information is always calculated.

This thesis is addressed to two strands of investigation: the questions of efficiency, and of expressivity.

An alternative paradigm is proposed, inspired by the Event Calculus, which considers only the information relevant to a given fluent's value. Current algorithms for working with $\mathcal{C}+$ employ propositional satisfaction solvers; that described in this thesis uses logic-programming throughout. Proofs of correctness are provided, and the correspondence with a variant of the event calculus is systematically investigated. (A consequence of this work is the provision of a transition system semantics for this variant of the Event Calculus.) We also compare the performance of our system, with the current implementation.

We would often like to be able constrain the behaviour of systems we represent depending on what has occurred in the distant past, yet in $\mathcal{C}+$ there is no convenient way of doing this. Accordingly, we make use of relations between $\mathcal{C}+$ and the formalism of 'non-monotonic causal theories' to broaden the expressivity of $\mathcal{C}+$ action descriptions, and present an updated semantics based on an interesting generalization of the standard transition systems. A proof of correctness for this semantics is given.

Finally, the standard query language for $\mathcal{C}+$ suffers in its expressivity, with many interesting statements about domains unable to be made; for instance, one cannot ask whether some fluent eventually holds. Accordingly, we investigate the possibility of using the technique of model-checking on transition systems defined using $\mathcal{C}+$, and show how these techniques can be used to verify that statements of standard temporal logics hold. Several different implementation routes are examined, executed and compared, including an interface with one representative, state-of-the-art model checker.

Acknowledgements

I would first like to express my gratitude to my supervisor, Marek Sergot, who provided the initial stimulus for many of the ideas contained in this thesis, and who guided their development and realization with much-appreciated and unfailing insight, care and humour. Our conversations about matters logical and non-logical have been sustenance without which this Ph.D. would certainly not have existed. For discussions about many points related to themes touched on, more or less directly, by my research, I also gladly thank Alexander Artikis.

Warm thanks to other friends in the Department of Computing: Irene Papatheodorou, William Heaven, and Matthew Smith.

Jessica Frazier read parts of the thesis very closely, at a late stage, finding numerous typographical errors and several less excusable howlers. For that, and for much else, I thank her.

Finally, I would also like to express my thanks to my parents, who encouraged me to pursue both the M.Sc. and the Ph.D. to which it led, and who have always been a source of help and love when I needed it.

Contents

List of Figures

Chapter 1

Introduction

Action languages are logical languages used for describing and reasoning about how the properties of a system change as a consequence of actions performed in the system. Current action languages provide a natural treatment for the inertia—the temporal persistence by default—of fluents; for the representation of concurrent actions; for making fluents take certain values by default and actions be performed a certain way by default; and for non-deterministic effects. An example of such a language is $\mathcal{C}+$ [GLL$^+$04], and the natural treatment $\mathcal{C}+$ affords for the features of domains just enumerated makes it one of the most useful and natural action languages currently available. Sets of formulas in $\mathcal{C}+$, known as *causal laws*, define labelled transition systems which encode the behaviour of the system represented.

Current methods for answering queries about domains formalized using this action language depend on an underlying mechanism of propositional satisfaction solving. One specifies the behaviour of a system, asserts certain known facts about when actions occur, or which properties of the system represented hold at which times, and the information is encoded and sent to a SAT-solver, which finds models which represent runs of the system along which the known facts are true. Crucially, this process *always* requires *complete* information about the whole run of the system to be computed.

Yet when the systems which we represent are large, this method can become very computationally expensive. One area in which we are interested in applying action languages (an area where the language features can be put to particularly good use) is that of *multi-agent systems*; these systems are often composed of large numbers of interacting components, where the interactions are complex, reflecting the scope of the component agents for behaviour which is nuanced and intelligent. In order to make $\mathcal{C}+$ more suitable for reasoning about the properties of such large, complex systems over long runs, we should try and find ways of making the answering of queries of $\mathcal{C}+$ domains much more efficient.

To that end, we have investigated how $\mathcal{C}+$ might be related to the Event Calculus, and how we might retain the useful graphical semantics and expressivity of the former whilst using a mode of more relevantly, goal-directed computation of the sort afforded by the latter. Top-down queries to variants of the Event Calculus expressed as logic programs make use of the laws expressing interaction amongst fluents and actions, to consider only that information which may be relevant to the truth of a fluent. We take the essence of this insight and apply

it to $\mathcal{C}+$, to give a logic-programmed version of $\mathcal{C}+$ which, for certain kinds of query, affords several advantages over the previous computational model.

Whilst one strand of investigation of this thesis has been that of efficiency, the other is that of expressivity. $\mathcal{C}+$ as it stands is limited in ways that have seemed to us both inconvenient and unnecessary; this affects both the causal laws of $\mathcal{C}+$ itself and the query language which is standardly used to express queries about the systems the laws define. Thus, in causal laws as they currently stand in $\mathcal{C}+$, one may express direct relationships between fluents at the same time, or else between fluents of one time and the immediately succeeding. Direct relationships between actions are only expressible when the actions occur at the same time. Yet runs through systems represented in $\mathcal{C}+$ can be defined in terms of an underlying formalism which makes no such restriction on the interactions between different times, where events and properties can have temporally distant effects on each other. We have lifted this expressivity of the underlying formalism up into $\mathcal{C}+$, and in so doing have enabled the concise representation of domains involving deadlines and other forms of distant interaction.

In addition to broadening the expressivity of causal laws, we have also investigated several possibilities of connecting $\mathcal{C}+$ to model-checking. This has been motivated by the desire to enable the verification that the systems we describe satisfy a broader class of properties than is now possible; the temporal logics which are used in the model-checkers we employ are much more expressive than the standard query language for $\mathcal{C}+$. Applications of model-checking to multi-agent systems have recently proliferated, and investigations made of the verification of epistemic, deontic, and other prominent features of agentive interactions. The suitability, in our eyes, of $\mathcal{C}+$ and extensions of it as a representation and specification language for multi-agent systems immediately makes appropriate a study of ways to model-check systems defined using its causal laws.

1.1 Collaborations and Contributions

A preliminary version of Chapter 4 of this thesis was published as [CS05]. Some of the background material in Chapter 2 has been adapted from that paper and [SC05a].

The thesis contributes to the study of formalisms for reasoning about action and change. The main original results are: a demonstration of how action descriptions of the language $\mathcal{C}+$ can be represented as logic programs (Theorem 3.10); an implementation of these ideas which supports several types of useful query and which has been shown to perform better than the current implementation on sample domains; a theorem relating a variant of the Event Calculus to action descriptions of $\mathcal{C}+$ (Theorem 3.14), which has as a corollary a theorem relating our logic-programmed version of $\mathcal{C}+$ to the version of the Event Calculus which inspired it (Theorem 3.15); a generalization of $\mathcal{C}+$ laws to accommodate temporally distant causation; a semantics for sets of those laws which is proved correct (Theorem 4.4); investigation of the interaction between our generalization and $n\mathcal{C}+$ [SC06], a language introducing deontic concepts into $\mathcal{C}+$; three implementations which enable model-checking for systems described using action descriptions of $\mathcal{C}+$; and an investigation of the performance of these three implementations.

1.2 Structure of the thesis

The structure of this thesis is as follows. In Chapter 2 we present the necessary background concepts, theorems and notations upon which we shall rely for the remaining chapters, and give a survey of related work. The bulk of this chapter is concerned with introducing the action language $\mathcal{C}+$, its syntax and semantics. We also describe a modification of $\mathcal{C}+$ which introduces deontic concepts into the language, enabling a partitioning of states and transitions into those which are deontically acceptable and not. We also rehearse the stable model semantics for logic programs with negation-by-failure, which will be the semantics we shall use for the logic programs, of various sorts, to which we refer elsewhere in the thesis. A brief overview of model-checking is provided, and we conclude with some references to approaches and work related to our own in the literature.

Chapter 3 shows how a significant subset of action descriptions of $\mathcal{C}+$ (a subset we call $\mathcal{EC}+$) can be represented concisely and correctly as logic programs, and presents various applications, theorems and implementation details in relation to this work. We begin by defining that subset of $\mathcal{C}+$ to which the work applies, and describe the syntax of its representation in Prolog. The axioms, inspired by those for the event calculus, which are used in answering queries of domains are explained, and a substantial result proves that stable models of our logic programs correspond to runs through the transition systems defined by the $\mathcal{C}+$ action description. We explain the process of checking our logic programs for a necessary kind of consistency, and relate the forms of query and other tasks supported by the implementation. Several examples are presented: one illustrative, and one benchmark example from the literature on reasoning about action and change. We describe techniques used to avoid recomputation in our programs, and prove several theorems relating $\mathcal{EC}+$ to a variant of the Event Calculus.

In Chapter 4 we relax the syntax of causal laws in order to enable reference to actions temporally distant from each other, and fluents more than one time-step distant. We show which families of causal theory the new action descriptions define, and demonstrate that a new kind of graphical semantics would be needed, as the labelled transition systems defined according to the standard semantics would fail to capture the behaviour of the system. We supply this new graphical semantics, show how to generate it from action descriptions, and prove this process of generation to be correct. Several illustrations of the increased expressivity we have introduced are given, and we also describe the relationship between $n\mathcal{C}+$ (a deontic variant of $\mathcal{C}+$) and our new laws, showing how a combined system is possible.

Chapter 5 turns from the causal laws used to define the behaviour of systems to the query languages used to express properties of the systems defined. The labelled transition systems of $\mathcal{C}+$ are related to the finite state machines upon which model checkers typically verify properties of temporal logics, and this relation is used when we present three different bridges between $\mathcal{C}+$ and model checking. The first adapts CCalc to enable bounded model checking of formulas of LTL; the second translates a subset of action descriptions of $\mathcal{C}+$ into SMV, the input language of a standard model checker, and thus enables formulas both of LTL and CTL to be checked on domains; and the third implementation removes the restrictions, passing a representation of the entire transition system to NuSMV, for LTL and CTL model checking again. We present experimental

comparisons of our three approaches.

Finally, Chapter 6 is the conclusion, in which we summarize what has been achieved and discuss directions for future work. Several Appendices contain the code used to represent sample domains from Chapter 3.

Chapter 2

Background and Related Work

Much of the work described in this thesis is based on the action language $\mathcal{C}+$ [GLL$^+$04] of Giunchiglia, Lifschitz, Lee, McCain and Turner, and a sound understanding of this language will be essential to the comprehension of what follows. Accordingly, we give a rigorous, if brief, introduction to the language here, with technical details which will be relevant in later chapters. Although the material presented can be read without prior acquaintance with $\mathcal{C}+$, the reader is advised to consult [GLL$^+$04] and related documents for a fuller and more leisurely account.

After presenting the syntax and semantics of $\mathcal{C}+$, and giving an account of the very closely related language of *causal theories* (see [GLL$^+$04] again, and also [MT97] for the original presentation), we will describe an extension of $\mathcal{C}+$ incorporating language features for the representation of deontic concepts: in particular, those of permission and obligation. This extension was originally called $(\mathcal{C}+)^{++}$ [Ser04], and was renamed in a recent paper [SC06] to $n\mathcal{C}+$. We move on to rehearse the *stable model semantics* for logic programs [GL88], which is the semantics we shall use throughout the thesis to prove several theorems equating formalizations of domains in different sorts of logic program, and between logic programs and other representations.

We then define a variant of the *event calculus* [KS86], an inspiration for the logic-programmed form of $\mathcal{C}+$ action descriptions which will be shown in Chapter 3. Before describing other work related to the content of the thesis, a short overview of model checking is given: in particular, *bounded model checking*, central to Chapter 5 of the present work.

2.1 The Action Language $\mathcal{C}+$

Action languages are logics for describing how a system behaves over time, as a consequence of actions performed within the system. $\mathcal{C}+$ is the most recent member of a family of action languages which began with $\mathcal{A}$ [GL93], and whose early history is surveyed in [GL98]. $\mathcal{C}+$ has a number of very appealing features, which recommended it to us as a starting-point for our developments.

- It provides a very natural treatment of inertia (default persistence), which gives intuitively desired results.
- It is easy to say many things in $\mathcal{C}+$: concurrent actions, default effects, ramifications, non-determinism, are all accommodated.
- After only a brief acquaintance with the language, one can write action descriptions which capture the intended behaviour.
- The underlying formalism that can be provided—that of causal theories—is simple: so in cases where there are complex interactions between causal laws, reference to this underlying formalism quickly resolves confusion.
- There is a semantics of labelled transition systems already in place. Since we are interested in making use of the bridge this affords to methods in other areas of AI, this is a significant advantage.

We will give an overview of the syntax of $\mathcal{C}+$; show how labelled transition systems are defined; describe the underlying framework of causal theories and the alternative route to transition systems this affords; present the algorithm used to find models of an important subclass of causal theories; sketch the current implementation; and give an outline of $n\mathcal{C}+$ [Ser04], an extension to $\mathcal{C}+$ incorporating deontic concepts.

2.1.1 Signatures and Causal Laws

First, the syntax of $\mathcal{C}+$. We begin with σ, a multi-valued, propositional signature. Members of σ are known as *constants*. σ is assumed to be partitioned into a set σ^f of *fluent constants* and a set σ^a of *action constants*. Further, the fluent constants are partitioned into those which are *simple* and those which are *statically determined*. We sometimes use σ^{smpl} to stand for the simple fluent constants, and σ^{stat} to stand for the statically determined fluent constants. And so:

$$\sigma \quad = \quad \sigma^f \cup \sigma^a \quad = \quad \sigma^{smpl} \cup \sigma^{stat} \cup \sigma^a.$$

Since many of our action descriptions will contain no statically determined fluent constants, we sometimes specify the fluent constants by writing simply $\sigma^f = X$, meaning that X is the set of simple fluent constants and σ^{stat} is empty.

For each constant $c \in \sigma$ there is a finite, non-empty set $dom(c)$, disjoint from σ and known as the *domain* of c. An *atom* of the signature is an expression $c{=}v$, where $c \in \sigma$ and $v \in dom(c)$. *Formulas* are constructed from the atoms using propositional connectives and a familiar syntax, with a *literal* as an expression A or $\neg A$, for atomic A. The expressions $\top$ and $\bot$ are connectives of zero arity, with the usual interpretation. A *Boolean* constant is one whose domain is the set of truth-values $\{\mathbf{t}, \mathbf{f}\}$, and a *Boolean* signature is, by extension, one all of whose constants are Boolean. If c is a Boolean constant, we often write c for $c{=}\mathbf{t}$, so that where our propositional signatures are restricted to be Boolean and we deal with no formula containing $\mathbf{f}$, we may reduce our syntax to that of standard propositional logic.

A *fluent formula* is a formula whose constants all belong to σ^f; an action formula is a formula which contains at least one action constant, and no fluent constants. Note that according to these definitions, $\bot$ and $\top$ are fluent formulas

but not action formulas. Disjointness of σ^f and σ^a forces disjointness of the set of fluent formulas (which we abbreviate to $fmla^f$) and the set of action formulas (abbreviated to $fmla^a$). The disparity in the definitions of $fmla^f$ and $fmla^a$ makes later definitions more compact.

An *interpretation* of a multi-valued propositional signature σ is a function mapping every constant c to some $v \in dom(c)$; an interpretation X is said to *satisfy* an atom $c{=}v$ if $X(c) = v$, and in this case one may write $X \models c{=}v$. Standard structural recursions over the propositional connectives apply, and where Γ is a set of formulas of our propositional signature, $X \models \Gamma$ expresses that $X \models c{=}v$, for every $c{=}v$ in Γ. We let the expression $\mathrm{I}(\sigma)$ stand for the set of interpretations of σ.

A *static law* is an expression of the form

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

where F and G are fluent formulas. These laws are similar to the *state constraints* which appear elsewhere in computer science; their meaning is that when the formula G is true in a state, then the formula F *is caused* to be true. An *action dynamic law* is an expression of the same form in which F is an action formula and G is a formula: an action dynamic law **caused** A **if** G means that when G is true in a state, then when the system evolves from that state, it must do so in a way which makes A true. A *fluent dynamic law* has the form

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

where F and G are fluent formulas and H is a formula, with the restriction that F must not contain statically determined fluents. Typically H is a combination $A \wedge H_1$, where A is an action formula and H_1 a fluent formula: the fluent dynamic law can then be understood as stating that when H is true in a state, and the system being modelled performs action A, then *if* G is true in the succeeding state, F must *also* be true there. (These informal glosses will be made precise soon.) *Causal laws* are static laws or dynamic laws, and an *action description* is a set of causal laws.

When writing causal laws, we will frequently omit the keyword **caused**, for the sake of concision.

An action description D is said to be *definite* when

- the head of every causal law of D is either an atom or $\bot$, and
- no atom is the head of infinitely many causal laws of D.

When an action description of $\mathcal{C}+$ is definite in this sense, then there is a straightforward method of finding runs through the transition system it defines, which we will present in Section 2.1.5. All of the examples we will mention in the current thesis are definite action descriptions.

For the purpose of illustration, consider the very simple action description having as its Boolean signature

$$\sigma^{smpl} = \{p\}, \qquad \sigma^a = \{a\},$$
$$\sigma^{stat} = \{q\}.$$

Thus, p and q are intended to represent properties of states, and a to represent an action, the performance of which may affect those properties. Let the laws of the action description be:

exogenous a
q **if** p,
$\neg q$ **if** $\neg p$,
p **if** $\top$ **after** a,
$\neg p$ **if** $\top$ **after** $\neg a$.

Anticipating ourselves somewhat by assuming that action descriptions of $\mathcal{C}+$ can be rendered graphically, we show the behaviour of our system in Figure 2.1. The first two (static) laws make q's value dependent on that of p. The second

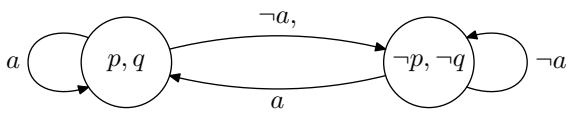

Figure 2.1: Transition system for a simple action description

two laws make the value of p change, depending on whether a is performed or not. We see that static laws can be used to describe how the effects of actions ramify.

2.1.2 Action Descriptions to Transition Systems

As we have hinted, every action description of $\mathcal{C}+$ defines a *labelled transition system*. In general, transition systems are graphs, whose vertices represent the states of some system, and where an edge between two vertices represents that the system may evolve from the one state to the other. The edges are typically called *transitions*, and where the transitions are labelled, the labels usually denote the performance of some action, or the execution of some computation, which effects the given transition between states. This sort of graphical structure is, of course, ubiquitous in computing and artificial intelligence.

The precise form of the transition systems with which we will deal in the current thesis will vary, usually as the language used to define them accrues special features. It is therefore difficult even to define an underlying template to which all labelled transition systems must conform. So we relativize, as follows.

Definition 2.1 A *transition system for* $\mathcal{C}+$ is a triple (S, L, R), where

- S is a set of *states*, sometimes called *vertices*;
- L is a set of *labels*;
- R is a set of *transitions*, $R \subseteq S \times L \times S$, sometimes called *edges*. ⌟

The purpose of this section is to describe how each action description D of $\mathcal{C}+$—with signature $\sigma^f \cup \sigma^a$—defines a labelled transition system $\mathcal{L}_D$, of the

form defined above. But we will first note that states will turn out to be interpretations of σ^a which satisfy certain constraints, and the set of labels will be the set $\mathrm{I}(\sigma^a)$. The definitions and theorems in this section are mostly taken from [Ser04].

Suppose we are given an action description D of $\mathcal{C}+$, with signature $\sigma^f \cup \sigma^a$.

Definition 2.2 We define:

$$\begin{aligned}
T_{\mathrm{static}}(s) &=_{def} \{F \mid F \textbf{ if } G \in D,\ F \in \mathit{fmla}^f,\ s \models G\},\\
E(s,e,s') &=_{def} \{F \mid F \textbf{ if } G \textbf{ after } H \in D,\ s \cup e \models H,\ s' \models G\},\\
A(s,e) &=_{def} \{A \mid A \textbf{ if } H \in D,\ A \in \mathit{fmla}^a,\ s \cup e \models H\},\\
\mathit{Simple}(s) &=_{def} \{c{=}v \mid c \in \sigma^{smpl},\ s \models c{=}v\},
\end{aligned}$$ ⌟

With these preliminary definitions, we can now define the transition systems defined by $\mathcal{C}+$ action descriptions. We first need to define our states.

Definition 2.3 Let D be an action description of $\mathcal{C}+$. An interpretation s of σ^f is a *state* of D iff

$$\{s\} = \{s' \in \mathrm{I}(\sigma^f) \mid s' \models T_{\mathrm{static}}(s) \cup \mathit{Simple}(s)\}.$$ ⌟

In words, s is a state when s satisfies the set of forumlas $T_{\mathrm{static}}(s) \cup \mathit{Simple}(s)$, and there is no other interpretation of σ^f which satisfies this set. We move on to the transitions defined by $\mathcal{C}+$ action descriptions.

Definition 2.4 Let D be an action description of $\mathcal{C}+$, with signature σ. Suppose $s, s' \in \mathrm{I}(\sigma^f)$, and $e \in \mathrm{I}(\sigma^a)$. We say that (s,e,s') is a *transition* of D iff s is a state (according to the preceding definition) and:

- $\{s'\} = \{s'' \in \mathrm{I}(\sigma^f) \mid s'' \models T_{\mathrm{static}}(s') \cup E(s,e,s')\}$;
- $\{e\} = \{e' \in \mathrm{I}(\sigma^a) \mid e' \models A(s,e)\}$. ⌟

Again, in words: (s,e,s') is a transition when s is a state; where s' satisfies $T_{\mathrm{static}}(s') \cup E(s,e,s')$ and no other interpretation does so; and where e satisfies $A(s,e)$ and no other interpretation in σ^a does so.

The transition systems defined by a $\mathcal{C}+$ action description have the form given above in Definition 2.1: the component S is the set of states defined by the action description D; the possible labels L are found in $\mathrm{I}(\sigma^a)$; and R is the set of transitions, as given in Definition 2.4.

Theorem 2.5 If (s,e,s') is a transition of an action description D, then s' is a state.

Proof: This is Proposition 4 of [Ser04]. ⌟

At a first glance the reason that these definitions take precisely the form they do is far from obvious, and they are apt to seem somewhat arbitrary. Matters become clearer when we examine the relationship of action descriptions of $\mathcal{C}+$ to causal theories.

2.1.3 Causal Theories

The language of causal theories ([MT97], [GLL$^+$04]) is a more general-purpose, non-monotonic formalism which can be seen as underlying the action language $\mathcal{C}+$. (The manner in which it is non-monotonic is not entirely straightforward: see [SC05a] for details.) Causal theories are very closely related to Reiter's Default Logic [Rei80], as Section 7 of [GLL$^+$04] shows, and it is partly their scope for a highly nuanced representation of default behaviour which brings causal theories close to $\mathcal{C}+$. It will be seen that $\mathcal{C}+$ action descriptions correspond to families of certain forms of causal theories; but this correspondence does not use the full expressivity of causal theories, which is something we exploit in Chapter 4 when we draw on more of their expressive resources to broaden $\mathcal{C}+$.

In causal theories we start, as with $\mathcal{C}+$, with a multi-valued propositional signature σ. In the language of causal theories, however, there is no distinction between fluent constants and action constants—members of σ are undifferentiated. Formulas are built up from atoms $c{=}v$ using standard propositional connectives, again including $\top$ and $\bot$ as connectives of zero arity.

A *causal rule* is an expression of the form

$$F \Leftarrow G,$$

where F and G are formulas of the underlying, multi-valued propositional signature. Such expressions are related to the (almost) natural language statement "if G, then the fact that F is caused"; perhaps they could better be paraphrased as "if G, then there is a reason for F to be true (and F *is* true)". A *causal theory* is a set of causal rules.

Now let Γ be a causal theory, and take X to be an interpretation of its underlying propositional signature. The *reduct* of Γ with respect to X is defined as

$$\Gamma^X \ =_{def} \ \{F \mid F \Leftarrow G \in \Gamma \text{ and } X \models G\}.$$

X is a model of the causal theory Γ, written $X \models_{\mathsf{C}} \Gamma$, if X is the unique model of Γ^X. (This uniqueness constraint is related to the role of singletons in Definition 2.4.)

For an illustration of the preceding definitions, consider the causal theory T_1, with underlying Boolean signature $\{p, q\}$:

$$p \Leftarrow q$$
$$q \Leftarrow q$$
$$\neg q \Leftarrow \neg q$$

There are clearly four possible interpretations of the signature:

$$X_1\colon\ p \mapsto \mathbf{t},\ q \mapsto \mathbf{t}$$
$$X_2\colon\ p \mapsto \mathbf{t},\ q \mapsto \mathbf{f}$$
$$X_3\colon\ p \mapsto \mathbf{f},\ q \mapsto \mathbf{t}$$
$$X_4\colon\ p \mapsto \mathbf{f},\ q \mapsto \mathbf{f}$$

and it is clear that

$$T_1^{X_1} = \{p, q\} \text{ whose only model is } X_1$$
$$T_1^{X_2} = \{\neg q\} \text{ which has two models}$$
$$T_1^{X_3} = \{p, q\} \text{ whose only model is } X_1 \neq X_3$$
$$T_1^{X_4} = \{\neg q\} \text{ which has two models}$$

In only one of these cases—that of X_1—is it true that the reduct of the causal theory with respect to the interpretation has that interpretation as its unique model. Thus $X_1 \models_{\mathsf{C}} T_1$ and X_1 is the only model of T_1.

For logical properties of causal theories, and relations to modal logic, see [SC05b] (or its expanded version [SC05a]); also see [EL06].

2.1.4 Action Descriptions to Causal Theories

Action descriptions of $\mathcal{C}+$ can be seen as shorthand for families of causal theories. The index set is the non-negative integers, which represents the time for which the system runs.

Thus to every action description D of $\mathcal{C}+$—with signature σ—and non-negative integer t, there corresponds a causal theory Γ_t^D. The signature of Γ_t^D contains the constants $c[i]$,[1] such that

- $i \in \{0, \ldots, t\}$ and $c \in \sigma^f$, or
- $i \in \{0, \ldots, t-1\}$ and $c \in \sigma^a$,

and the domains of such constants $c[i]$ are kept identical to those of their constituents c in the signature of the action description. Where σ is the signature of the $\mathcal{C}+$ action description D, we will let σ_m denote the signature of the causal theory Γ_m^D. The expression $F[i]$, where F is a formula, denotes the result of inserting $[i]$ after every occurrence of a constant in F. The causal rules of Γ_t^D are:

$$F[i] \Leftarrow G[i],$$

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

$$F[i+1] \Leftarrow G[i+1] \wedge H[i],$$

for every fluent dynamic law in D and every $i \in \{0, \ldots, t-1\}$; and

$$c[0]{=}v \Leftarrow c[0]{=}v,$$

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

We have already defined the labelled transition systems which are determined by $\mathcal{C}+$ action descriptions, in a way which did not depend at all on the formalism of causal theories. The same transition systems can be defined much more succinctly using causal theories, however, and it is often much more easy when trying to imagine the systems defined by action descriptions, to think in terms of the causal-theoretic definitions rather than those given in Section 2.1.2.

[1] These are written as $i : k$ in [GLL+04]; we find the current notation easier to read.

In the current context we will identify interpretations of the underlying propositional signature of D with the sets of atoms they satisfy. Where i is a non-negative integer and s an interpretation, we can write $s[i]$ for the result of suffixing $[i]$ to every constant of an atom made true by the interpretation (in symbols, $\{c[i]{=}v \mid s \models c{=}v\}$).

We trespass on our previous definitions:

Definition 2.6 Let D be an action description of $\mathcal{C}+$, with signature σ.

- A *state* is any $s \in \mathrm{I}(\sigma^f)$, such that $s[0] \models_{\mathsf{C}} \Gamma_0^D$;
- a *transition* is any triple $(s, e, s') \in \mathrm{I}(\sigma^f) \times \mathrm{I}(\sigma^a) \times \mathrm{I}(\sigma^f)$ such that

$$s[0] \cup e[0] \cup s'[1] \models_{\mathsf{C}} \Gamma_1^D.$$ ⌟

According to this definition, the component S of the labelled transition systems defined by $\mathcal{C}+$ action descriptions is the set

$$\{s \mid s \in \mathrm{I}(\sigma^f),\ s[0] \models \Gamma_0^D\};$$

the set of labels L is $\mathrm{I}(\sigma^a)$ as before; and the set of edges is the set

$$\{(s, e, s') \mid s, s' \in \mathrm{I}(\sigma^f),\ e \in \mathrm{I}(\sigma^a),\ s[0] \cup e[0] \cup s'[1] \models \Gamma_1^D\}.$$

Theorem 2.7 Let D be an action description. Then for any transition (s, e, s'), s' is a state, where transitions and states are both defined according to Definition 2.6.

Proof: This is Proposition 7 of [GLL$^+$04]. ⌟

We now have two alternative definitions for the labelled transition systems defined by our action descriptions. The following theorem shows they coincide.

Theorem 2.8 Let D be an action description of $\mathcal{C}+$, with signature σ. Then:

(i) s is a state of D according to Definition 2.3 iff s is a state of D according to Definition 2.6;

(ii) (s, e, s') is a transition of D according to Definition 2.4 iff (s, e, s') is a transition of D according to Definition 2.6.

In other words, Definitions 2.3, 2.4 and 2.6 coincide.

Proof: This is essentially Theorem 9 of [Ser04], though the proof has to be changed slightly in order to accommodate the fact that the exogeneity of an action constant is not, in the most recent versions of $\mathcal{C}+$, a feature of the signature, but is signalled by the presence of action dynamic laws **caused** $a{=}v$ **if** $c{=}v$. ⌟

Let Γ_t^D be the causal theory generated from the action description D and non-negative integer t as described above. Let $s_0, \ldots, s_t$ be interpretations of σ^f and $e_0, \ldots, e_{t-1}$ be interpretations of σ^a. Then using the notation above, we can represent interpretations of the signature of Γ_t^D in the form

$$s_0[0] \cup e_0[0] \cup s_1[1] \cup e_1[1] \cup \cdots \cup e_{t-1}[t-1] \cup s_t[t] \tag{2.1}$$

The following result holds.

Theorem 2.9 An interpretation of the signature of Γ_t^D is a model of Γ_t^D iff each triple (s_i, e_i, s_{i+1}), for $0 \leqslant i < t$, is a transition.

Proof: Proposition 8 of [GLL$^+$04]. ⌟

Let D be an action description of $\mathcal{C}+$. A *run* of length t through this transition system is defined to be a sequence

$$(s_0, e_0, s_1, e_1, \ldots, e_{t-1}, s_t) \tag{2.2}$$

such that all triples (s_i, e_i, s_{i+1}), for $0 \leqslant i < t$, are members of the transition system.

Theorem 2.10 Let D be an action description and t any non-negative integer. Then the sequence (2.2) is a run of the transition system iff the interpretation (2.1) is a model of the causal theory Γ_t^D.

Proof: First, assume we have a run of the transition system of length t. Then every triple (s_i, e_i, s_{i+1}), for $0 \leqslant i < t$, is a transition, and so by Theorem 2.9 the interpretation (2.1) is a model of Γ_t^D.
Alternately, suppose that (2.1) is a model of the causal theory Γ_t^D, Then clearly, each triple (s_i, e_i, s_{i+1}), for $0 \leqslant i < t$, is a transition, and so the sequence (2.2) is a run of the transition system defined by D. ⌟

2.1.5 Definiteness and Completion

A causal theory Γ is said to be *definite* when:

- the head of every causal rule is either an atom or $\bot$, and
- no atom is the head of infinitely many causal rules.

Clearly this definition is related to that of the definiteness of an action description of $\mathcal{C}+$, given in Section 2.1.1. Indeed, it turns out that when an action description D is definite, then the causal theories Γ_t^D, for $0 \leqslant t$, are also definite.

By a process similar to the Clark completion used in the semantics of logic programming [Cla78], one can reduce the problem of finding models of a definite causal theory, to that of finding the models of a set of formulas of propositional logic. The latter problem can be shipped out to a propositional satisfaction solver. So, in the light of Theorem 2.10, we have a means of finding runs of length $t \leqslant 0$ through the transition system defined by a definite action description of $\mathcal{C}+$.

Let Γ be a definite causal theory, of which we wish to find the completion, and let Γ have signature σ. An atom $c{=}v$ of σ is said to be *trivial* when $dom(c) = \{v\}$: in this case any interpretation of σ must assign to c the value v. (We also call the constant c trivial in this case.) We let *Trivial*(σ) be the set of atoms $c{=}v$ of σ such that c is trivial. For each non-trivial atom $c{=}v$ of σ, the *completion formula* is:

$$c{=}v \equiv G_1 \vee \cdots \vee G_n,$$

where

$$c{=}v \Leftarrow G_1, \ldots, c{=}v \Leftarrow G_n$$

are all the rules in Γ with head $c{=}v$. The *completion* of Γ is the set of all completion formulas of all non-trivial atoms in σ, together with all formulas $\neg F$ for each rule

$$\bot \Leftarrow F$$

in Γ. We let $comp(\Gamma)$ be the completion of Γ, so that in symbols:

$$\begin{aligned} comp(\Gamma) =_{def}\ & \{c{=}v \equiv G_1 \vee \cdots \vee G_n \mid c{=}v \in \mathit{Trivial}(\sigma), \\ & \qquad \forall G(c{=}v \Leftarrow G \in \Gamma \leftrightarrow \exists i(0 \leqslant i \leqslant n \wedge G = G_i))\} \\ & \cup \{\neg F \mid \bot \Leftarrow F \in \Gamma\} \end{aligned}$$

Theorem 2.11 The models of a definite causal theory Γ are precisely the models of its completion.

Proof: Proposition 6 of [GLL$^+$04]. ⌟

2.1.6 Abbreviations

When working with $\mathcal{C}+$ it is best to make use of the many abbreviations which have been introduced, to enable the compact representation of system behaviour. We tabulate some of the most useful here; the reader is referred to Appendix B of [GLL$^+$04] for the full list.

We divide the abbreviations according to whether they apply to static, fluent dynamic, or action dynamic rules.

Abbreviation	Longhand
default F	F **if** F
default F **if** G	F **if** $F \wedge G$
F	F **if** $\top$
constraint F	$\bot$ **if** $\neg F$
F **after** H	F **if** $\top$ **after** H
inertial c $\quad (c \in \sigma^f)$	$\{c{=}v$ **if** $c{=}v$ **after** $c{=}v \mid v \in dom(c)\}$
nonexecutable F	$\bot$ **if** $\top$ **after** F
nonexecutable F **if** G	$\bot$ **if** $\top$ **after** $F \wedge G$
A **may cause** F **if** H $\quad (F \in fmla^f)$	F **if** F **after** $H \wedge A$
A **causes** F **if** H $\quad (A \in fmla^a)$	F **if** $\top$ **after** $H \wedge A$
constraint F **after** H	$\bot$ **if** $\neg F$ **after** H
rigid c $\quad (c \in \sigma^f)$	$\{\bot$ **if** $\neg c{=}v$ **after** $c{=}v \mid v \in dom(c)\}$
always F	$\bot$ **if** $\top$ **after** $\neg F$
default F **if** G **after** H	F **if** $F \wedge G$ **after** H
A	A **if** $\top$
A **causes** B **if** H $\quad (A, B \in fmla^a)$	B **if** $H \wedge A$
A **may cause** B **if** H	B **if** $B \wedge H \wedge A$

2.1.7 Example

To illustrate some of the expressive capabilities of $\mathcal{C}+$, we now include an example, the 'Farmyard Resurrection' domain.

At its core, this is a version of the Yale Shooting Problem [HM87]. There are two agents, *Bill* and *Turkey*, who can be either alive or dead, and either smiling or not; the dead do not smile. Bill and Turkey may walk between three locations: the barn, field, and house. A gun, which may be loaded or not, can point at any of the three locations, or nowhere (represented as *none*). If the gun is aimed at one of the locations, that location becomes the *target*; loading the gun causes the target to revert to *none*. If the gun is shot whilst pointing at a location, anything alive there is killed. The performance of a miracle on either Bill or Turkey makes them smile (and as a ramification, brings them back to life).

Our signature will be as follows (x ranges over $\{Bill, Turkey\}$):

$$\begin{aligned}\sigma^{smpl} &= \{alive(x), loaded, smiling(x), target, loc(x)\}\\ \sigma^{stat} &= \{\}\\ \sigma^{a} &= \{aim, miracle(x), load, shoot, walk(x)\}\end{aligned}$$

All constants are Boolean (domain $\{\mathbf{t}, \mathbf{f}\}$), except:

$$\begin{aligned}dom(loc) &= \{barn, field, house\}\\ dom(target) &= \{barn, field, house, none\}\\ dom(aim) = dom(walk) &= \{barn, field, house, \mathbf{f}\}\end{aligned}$$

Here is the action description. x ranges over $\{Bill, Turkey\}$ and l ranges over $\{barn, field, house\}$.

inertial c	(for every $c \in \sigma^f$)
exogenous a	(for every $a \in \sigma^a$)
nonexecutable $walk(x)=l$ **if** $loc(x)=l$	
nonexecutable $walk(x)=l$ **if** $\neg alive(x)$	
$alive(x)$ **if** $smiling(x)$	
$\neg smiling(x)$ **if** $\neg alive(x)$	
$shoot$ **causes** $\neg alive(x)$ **if** $loaded \wedge target{=}l \wedge loc(x){=}l$	
$shoot$ **causes** $\neg loaded$ **if** $loaded$	
$load$ **causes** $loaded$	
$walk(x)=l$ **causes** $loc(x)=l$	
$miracle(x)$ **causes** $smiling(x)$ **if** $\neg alive(x)$	
$aim=l$ **causes** $target=l$	
$load$ **causes** $target=none$	

The **inertial** law at the top expresses our requirement that values of fluent constants should persist through time, unless there is a reason for them to change. The **nonexecutable** laws place constraints on when agents may walk: one cannot walk to where one is already; to walk one must be alive.

Two static laws follow, which guarantee that only those who are alive may smile. The second static law is *not* redundant: in general one cannot take the contrapositive of causal laws and keep the same defined transition system.

The fluent dynamic laws express the various effects of performing actions, and should be more or less self-explanatory. Actions which are quite complex may be performed in this system: for example, the gun may be loaded as the Turkey walks into the barn; or Bill may move to the field, as the Turkey moves from there into the house: concurrency is easily supported, and the effects of concurrent actions work out as one would wish.

In the absence of a law

$$\textbf{nonexecutable}\ \mathit{aim}{=}l \wedge \mathit{load},$$

it might be thought that the two last fluent dynamic laws are inconsistent. In reality, the effect is simply to remove edges e from the transition system such that

$$e \models \mathit{aim}{=}l \wedge \mathit{load}.$$

Clearly, all transitions (s, e, s') are such that:

$$\begin{aligned}&\big((s \models \mathit{target}{=}l \wedge \mathit{loc}(x){=}l \wedge \mathit{loaded}) \wedge\\ &\ (e \models \mathit{walk}(x){=}\mathbf{f} \wedge \mathit{shoot})\big)\\ &\qquad\rightarrow\\ &s' \models \neg\mathit{alive}(x)\end{aligned}$$

Investigation shows that it is *not* true that all runs $(s_0, e_0, \ldots, s_n)$ such that

$$\begin{aligned}s_0 &\models \mathit{target}{=}l \wedge \mathit{loc}(x){=}l \wedge \mathit{loaded}\\ e_i &\models \mathit{walk}(x){=}\mathbf{f} \wedge \mathit{aim}{=}\mathbf{f} \wedge \neg\mathit{load} \qquad (0 \leqslant i < n)\\ e_{n-1} &\models \mathit{shoot}\end{aligned}$$

must have $s_n \models \neg\mathit{alive}(x)$, as one might expect. For if $n > 1$, then we could have

$$e_0 \models \mathit{shoot} \qquad \text{and} \qquad e_1 \models \mathit{miracle}(x) \wedge \mathit{shoot}(x)$$

in which case $s_n \models \mathit{alive}(x)$, as a simple derivation, or experimentation with CCalc, will show. We do however have the weaker property that when a run $(s_0, e_0, \ldots, s_n)$ satisfies

$$\begin{aligned}s_0 &\models \mathit{target}{=}l \wedge \mathit{loc}(x){=}l \wedge \mathit{loaded}\\ e_i &\models \mathit{walk}(x){=}\mathbf{f} \wedge \mathit{aim}{=}\mathbf{f} \wedge \neg\mathit{load} \wedge \neg\mathit{miracle}(x) \qquad (0 \leqslant i < n)\\ e_{n-1} &\models \mathit{shoot}\end{aligned}$$

then $s_n \models \neg\mathit{alive}(x)$. This is to say that the problem which is highlighted in the 'Yale Shooting' domain is treated satisfactorily by this formalization in $\mathcal{C}+$.

2.1.8 Queries

The current implementation of $\mathcal{C}+$, CCalc, which we will describe in Section 2.1.9, allows one to specify action descriptions in $\mathcal{C}+$ and then to query the domains described. $\mathcal{C}+$ itself is not a query language: laws of $\mathcal{C}+$, when conjoined to an action description, simplify modify the structure of the labelled transition system defined, and are not themselves evaluated on that transition system. This means that a separate query language is needed in which

to express statements about the systems defined by causal laws, and CCalc currently accepts queries stated in the language we give below. (The authors of [GLL+04], though they describe the nature of the query language and how it is implemented, do not formalize it.)

For each query, the user specifies which lengths of paths through the transition system he wishes to consider, together with a set of fluent and action atoms indexed by times in a way consistent with the signature: the intention is to find the shortest paths of the length specified which satisfy the atoms which form the substance of the query. Since the length of the path may vary, a special time-index is provided to index the maximum time—thus making the query language very suitable for planning tasks (amongst others). We will render this special time-index as *max*. Additionally, each query is labelled with a unique identifier, a natural number.

Queries can therefore be considered as triples (L, T, N), where

- $L \in \mathbb{N}$ is a unique identifier;
- T is $[t_{min}, t_{max}]$, where this denotes an interval of $\mathbb{N}$ and $t_{min} \leqslant t_{max}$, or T is $[t, \infty)$, $t \in \mathbb{N}$;
- A is a set of atoms $c[i]{=}v$, where $i \in T$, and if $c \in \sigma^a$ and $T = [t_{min}, t_{max}]$, then $i < t_{max}$; or else i is *max* and $c \in \sigma^f$.

Given this language, planning queries would typically take the form

$$(n, T, \{c_0[0]{=}v_0, \ldots, c_m[0]{=}v_m, c'_0[max]{=}v'_0, \ldots c'_n[max]{=}v'_n\}),$$

where the initial state s_0 and goal state $s_{t_{max}}$ should satisfy

$$s_0 \models \bigwedge_{i=0}^{m} c_i{=}v_i \quad \text{and} \quad s_{t_{max}} \models \bigwedge_{i=0}^{n} c'_i{=}v'_i;$$

and 'postdiction' queries should take the form

$$(n, T, \{c_0[0]{=}v_0, \ldots, c_m[0]{=}v_m\})$$

with the initial state having to make the $c_i{=}v_i$ true, as before.

An atom $c[i]{=}v$ is true of a path $(s_0, e_0, \ldots, s_n)$ if $i < n$ and $s_i \cup e_i \models c{=}v$, or if $i = n$ and $s_i \models c{=}v$. An expression of the form $c[max]{=}v$ is true of a path when $s_n \models c{=}v$. A query (L, T, N) is true of a path π when the length of π is in T and π satisfies all members of N.

Clearly, the query language supported by CCalc, of which we have given a brief account here, is only one of many possible formal languages which may be evaluated on structures defined by $\mathcal{C}+$ action descriptions. The query language we have just described is evaluated on paths through the transition systems $\mathcal{L}_D$ of an action description D. We could also evaluate a number of temporal logics over such structures (a fact we rely on in Chapter 5, when we connect model-checking and temporal logics to $\mathcal{C}+$), or connect action descriptions to automata theory and process algebra (possible lines of future research).

2.1.9 Current Implementation

The system CCalc, maintained by people at the University of Texas, supports a wide range of tasks relating to $\mathcal{C}+$ and the language of causal theories.

Action descriptions of $\mathcal{C}+$ are placed in source files, with the signature of the language able to be defined using various time-saving shorthands. The abbreviations given above in Section 2.1.6 can be used, as well those remaining in [GLL+04]. Users may also submit queries expressed in the language defined in Section 2.1.8, so that CCalc can find paths of minimal length through $\mathcal{L}_D$ which satisfy them.

The signature, action description and queries are loaded into a Prolog interface which is running CCalc. The action description D is converted to the causal theory Γ_1^D, $comp(\Gamma_1^D)$ is calculated, and the resulting set of propositional formulas is converted into conjunctive normal form (CNF). Suppose we have some query (L, T, N). CCalc takes each $t \in T$ in increasing order, making a representation of a path of length t through the transition system by 'shifting' the clauses derived from $comp(\Gamma_1^D)$.[2] Then information about the query is added to these clauses: the set $Q_{T,N}$, which includes

- every $c[i]{=}v \in N$, with $i \in \mathbb{N}$;
- the atom $c[t]{=}v$, for every $c[max]{=}v \in N$.

The resulting set of formulas, which contains information about the transition system and information about the query, and can be written

$$cnf(comp(\Gamma_t^D) \cup Q_{T,N}),$$

is passed to a propositional satisfaction solver. Models of the propositional clauses are also models of the causal theory Γ_t^D. When CCalc finds a non-negative integer $t \in T$ such that there is a model of $cnf(comp(\Gamma_t^D) \cup Q_{T,N})$, then it outputs all such models. They correspond to the shortest paths through the transition system $\mathcal{L}_D$ which satisfy the constraints represented in the query.

Note that to solve any query, CCalc must find out the values of all fluents along a path through the transition system. So, when one is interested in long histories of complex systems, computations using CCalc quickly become infeasible. This failing signals the opportunity we exploit in Chapter 3.

2.2 The Red and the Green

To use a language such as $\mathcal{C}+$ for reasoning about certain kinds of multi-agent system, where heterogeneity of the agents and openness of the group of agents may bring with it agentive behaviour which is not in conformity to the protocol of the system, it frequently becomes necessary to introduce deontic concepts into

[2] This process is simply a matter of repeatedly incrementing the time-stamps in these clauses. It relies, at bottom, on the fact that $\Gamma_{k+1}^D = (\Gamma_k^D \cup \Gamma_*^D)$ for all k, where Γ_*^D is simply $(\Gamma_1^D - \Gamma_0^D)$ with the time-stamps incremented by k. Thus letting $(\Gamma_t^D)[{+}k]$ denote the causal theory obtained from Γ_t^D by changing any constant $c[i]$ occurring in Γ_t^D to a constant $c[i+k]$, we have that

$$\Gamma_k^D = \Gamma_0^D \cup \bigcup_{i=0}^{k-1} (\Gamma_1^D - \Gamma_0^D)[{+}i].$$

the language. Then protocols may be expressed, broken, and the concomitant violation represented. Such an extension of the basic language has been provided in [Ser04], where it was called $(\mathcal{C}+)^{++}$; an updated version of that subset of the extension in which we are interested was described in [SC06], and was renamed $n\mathcal{C}+$; we give a brief account here.

$n\mathcal{C}+$ provides a means of specifying what the permitted or acceptable states and transitions of the system are. We add to $\mathcal{C}+$ *permission laws*:

- a *state permission law* has the form

  $$n: \ \textbf{not-permitted}\ F\ \textbf{if}\ G,$$

  for $F, G \in \mathit{fmla}^f$, and where n is an (optional) identifier;

- an *action permission law* has the form

  $$n: \ \textbf{not-permitted}\ A\ \textbf{if}\ G,$$

  where $A \in \mathit{fmla}^a$, $G \in \mathit{fmla}^f$, and with n as before.

The significance of these laws should be intuitively obvious: a state permission law of the form given expresses that whenever a state satisfies G, then if F is also true in the state, something has gone wrong, the system is in a state of violation: in these circumstances, F is not permitted. An action permission law of the form given qualifies the deontic status of transitions: where G is true in a state, then the action A is bad, not permitted. These accounts will be made firmer.

Structure is added to the labelled transition systems defined by our action descriptions, to represent the information held in the permission laws.

Definition 2.12 A *transition system for* $n\mathcal{C}+$ is a tuple $(S, L, R, S_{\mathrm{g}}, R_{\mathrm{g}})$, where (S, L, R) is a transition system for $\mathcal{C}+$ (as given in Definition 2.1), and

- $S_{\mathrm{g}} \subseteq S$ is the set of *green states*, understood as those which are permitted, acceptable, ideal, deontically ideal, legal, etc.;
- $R_{\mathrm{g}} \subseteq R$ is the set of *green transitions*, which maybe understood as the transitions which are good, permitted, and so on.

We sometimes also refer to the complements of the green states and transitions:

- $S_{\mathrm{r}} \subseteq S$, where $S_{\mathrm{r}} = S - S_{\mathrm{g}}$, is the set of *red states*;
- $R_{\mathrm{r}} \subseteq R$, where $R_{\mathrm{r}} = R - R_{\mathrm{g}}$, is the set of *red transitions*. ⌟

Let D be an action description of $n\mathcal{C}+$ (there is no change to the signatures). The sets S, L and R are defined exactly as for $\mathcal{C}+$, using that subset of the laws of D which do not contain the keyword **not-permitted**. The permission laws of $n\mathcal{C}+$ action descriptions do not alter the structure of the defined transition system, but are used to determine the sets S_{red} and R_{red}.

Definition 2.13 Let D be an action description of $n\mathcal{C}+$, with signature σ. Let D_{perm} be the subset of permission laws of D. The transition system defined by D is the tuple $(S, L, R, S_{\mathrm{g}}, R_{\mathrm{g}})$, where S is the set of states, R the set of

transitions, defined by $D - D_{perm}$ according to Definitions 2.3 and 2.4; where L is $\mathrm{I}(\sigma^a)$, and where:

$$\begin{aligned}
S_{\mathrm{r}} &=_{def} \{s \in S \mid \exists \textbf{ not-permitted } F \textbf{ if } G \in D \ (s \models F \wedge G)\},\\
S_{\mathrm{g}} &=_{def} S - S_{\mathrm{r}}\\
R_{\mathrm{r}} &=_{def} \{(s,e,s') \in R \mid \exists \textbf{ not-permitted } A \textbf{ if } G \in D \ (s \cup e \models G,\ e \models A)\}\\
&\quad \cup \{(s,e,s') \in R \mid s \in S_{\mathrm{g}},\ s' \notin S_{\mathrm{g}}\}\\
R_{\mathrm{g}} &=_{def} R - R_{\mathrm{r}}
\end{aligned}$$ ⌟

So, adding permission laws to a $\mathcal{C}+$ action description effects a 'colouring' of the transition system, according to which states and transitions are shaded red or green. Any state s such that $s \models F \wedge G$ for some state permission law

$$\textbf{not-permitted } F \textbf{ if } G$$

is coloured red; all other states are green by default. Then, any transition (s, e, s') such that there is an action permission law

$$\textbf{not-permitted } A \textbf{ if } G$$

with $s \cup e \models G$ and $e \models A$ is coloured red. All other transitions are coloured red, except where this would contravene the so-called 'green-green-green' constraint (*ggg*, for short): for all transitions (s, e, s'), if $s \in S_{\mathrm{g}}$ and $e \in R_{\mathrm{g}}$, then $s' \in S_{\mathrm{g}}$. The rationale behind *ggg* is that whenever a system is in a permitted state, and performs an action which violates no deontic laws, the state which results from such a transition should also be permitted. (For more remarks on this justification, see [SC06].)

We can translate action descriptions of $n\mathcal{C}+$ into the language of causal theories, as follows. Let D be an action description and t a non-negative integer. The translation of the $\mathcal{C}+$-component, $D - D_{perm}$, proceeds as normal. For the permission laws, we introduce into the signature of Γ_t^D two new constants *status* and *trans*, both having domain $\{green, red\}$. For every state permission law **not-permitted** F **if** G in D_{perm}, we include the causal rules

$$status[i]{=}red \Leftarrow F[i] \wedge G[i],$$

for all i with $0 \leqslant i \leqslant t$. To make states green by default, we include the rules

$$status[i]{=}green \Leftarrow status[i]{=}green,$$

for all i with $0 \leqslant i \leqslant t$. For a law **not-permitted** A **if** G, we include the causal rules

$$trans[i]{=}red \Leftarrow A[i] \wedge G[i],$$

for all i with $0 \leqslant i < t$. We include rules

$$trans[i]{=}green \Leftarrow trans[i]{=}green,$$

for all i such that $0 \leqslant i < t$ to make transitions green by default, and finally, to enforce the green-green-green constraint, we include

$$trans[i]{=}red \Leftarrow status[i+1]{=}red \wedge status[i]{=}green,$$

for all i with $0 \leqslant i < t$.

Let D be an action description of $n\mathcal{C}+$, and $(S, L, R, S_{\mathrm{g}}, R_{\mathrm{g}})$ the defined labelled transition system. We define the function $\mathit{colour} : S \cup R \to \{\mathit{red}, \mathit{green}\}$ by:

$$\mathit{colour}(x) = \begin{cases} \mathit{green}, & \text{if } x \in S_{\mathrm{g}} \text{ or } x \in R_{\mathrm{g}}; \\ \mathit{green}, & \text{otherwise.} \end{cases}$$

The following result shows that there is a correspondence between models of Γ_t^D, for an action description D of $n\mathcal{C}+$, and runs through the transition system defined by D, in such a way that the constants $\mathit{status}[i]$ and $\mathit{trans}[i]$ encode details of the colouring of states and transitions.

Theorem 2.14 Let D be an action description of $n\mathcal{C}+$, with signature σ. Then, $(s_0, e_0, \ldots, s_t)$ is a path through the transition system defined by D iff

$$(s_0[0] \cup \{\mathit{status}[0]{=}\mathit{colour}(s_0)\}) \cup (e_0[0] \cup \{\mathit{trans}[0]{=}\mathit{colour}(e_0)\}) \cup \cdots \\ \cdots \cup (s_t[t] \cup \{\mathit{status}[t]{=}\mathit{colour}(s_t)\})$$

is a model of the causal theory Γ_t^D (signature $\sigma_t \cup \{\mathit{status}[i], \mathit{trans}[i'] \mid (0 \leqslant i \leqslant t) \wedge (0 \leqslant i' < t)\}$).

Proof: By induction on t.

(Base case: $t = 0$.) We want to show that s_0 is a state if and only if $s_0[0] \cup \{\mathit{status}[0]{=}\mathit{colour}(s_0)\}$ is a model of Γ_0^D. First assume that s_0 is a state. Then clearly $s_0[0] \models_{\mathsf{C}} \Gamma_0^{D-D_{perm}}$. If $s_0 \in S_{\mathrm{r}}$, then there is a state permission law

$$\textbf{not-permitted } F \textbf{ if } G \tag{2.3}$$

(ignoring the identifier) in D with $s_0 \models F \wedge G$. But then as

$$\mathit{status}[0]{=}\mathit{red} \Leftarrow F[0] \wedge G[0]$$

is in Γ_0^D, then $(\Gamma_0^D)^{s_0[0] \cup I}$ must contain $\mathit{status}[0]{=}\mathit{red}$ (where I is either the set $\{\mathit{status}[0]{=}\mathit{red}\}$ or $\{\mathit{status}[0]{=}\mathit{green}\}$, interpreting the special constant $\mathit{status}[0]$). Thus clearly $s_0[0] \cup \{\mathit{status}[0]{=}\mathit{red}\}$ is a model of Γ_0^D. If, instead, $s_0 \in S_{\mathrm{g}}$, then there is no law (2.3) in D_{perm}, and so there is no rule in Γ_0^D with $\mathit{status}[0]{=}\mathit{red}$ as its head. As

$$\mathit{status}[0]{=}\mathit{green} \Leftarrow \mathit{status}[0]{=}\mathit{green}$$

is in Γ_0^D, then $s_0[0] \cup \{\mathit{status}[0]{=}\mathit{green}\}$ is a model of Γ_0^D. Either way, then, $s_0[0] \cup \{\mathit{status}[0]{=}\mathit{colour}(s_0)\}$ is a model of Γ_0^D.
Alternately, suppose that $(s_0[0] \cup \{\mathit{status}[0]{=}\mathit{colour}(s_0)\}) \in \mathit{models}(\Gamma_0^D)$. Then as there is no rule in Γ_0^D with $c[0]{=}v$ (for $c \neq \mathit{status}$) as its head and the constant status appearing in its body, it follows easily that $s_0[0]$ is a model of $\Gamma_0^{D-D_{perm}}$, and thus s_0 is a state, i.e. a run of length 0.

(Inductive step: assume true for $t = k$, prove for $t = k + 1$.) First assume the result for $t = k$. Now let $(s_0, e_0, \ldots, s_k, e_k, s_{k+1})$ be a run through the transition system defined by D. Then so is $(s_0, e_0, \ldots, s_k)$, and so by the inductive hypothesis

$$(s_0[0] \cup \{\mathit{status}[0]{=}\mathit{colour}(s_0)\}) \cup \cdots \cup (s_k[k] \cup \{\mathit{status}[k]{=}\mathit{colour}(s_k)\})$$

is a model of Γ^D_k (we will call this model M). By Theorem 2.10, we have that $(s_0, e_0, \ldots, s_k, e_k, s_{k+1})$ is a model of $\Gamma^{D-D_{perm}}_{k+1}$. Let M_{k+1} be

$$\begin{aligned} M_k \;\cup\; & (e_k[k] \cup \{\mathit{trans}[k]{=}\mathit{colour}(e_k)\}) \\ \cup\; & (s_{k+1}[k+1] \cup \{\mathit{status}[k+1]{=}\mathit{colour}(s_{k+1})\}), \end{aligned}$$

so that M_{k+1} is an interpretation of the signature of Γ^D_{k+1}. Then a case analysis of the possibilities for the colours of e_k and s_{k+1} shows that $(\Gamma^D_{k+1})^{M_{k+1}}$ must contain $\mathit{trans}[k]{=}x$ iff $\mathit{colour}(e_k) = x$, and must contain $\mathit{status}[k+1]{=}x$ iff $\mathit{colour}(s_{k+1}) = x$. Thus M_{k+1} is the only model of $(\Gamma^D_{k+1})^{M_{k+1}}$, and so $M_{k+1} \models_{\mathsf{C}} \Gamma^D_{k+1}$.
For the other direction, assume

$$\begin{aligned} & (s_0[0] \cup \{\mathit{status}[0]{=}\mathit{colour}(s_0)\}) \cup \cdots \\ & \cdots \cup (s_{k+1}[k+1] \cup \{\mathit{status}[k+1]{=}\mathit{colour}(s_{k+1})\}) \end{aligned}$$

be a model of Γ^D_{k+1}. Then $(s_0[0], e_0[0], \ldots, s_k[k], e_k[k], s_{k+1}[k+1])$ is a model of $\Gamma^{D-D_{perm}}_{k+1}$, essentially as no causal rule in $\Gamma^{D-D_{perm}}$ has a head $c[i]{=}v$ for $c \in \sigma$, but a body containing a constant $\mathit{status}[j]$ or $\mathit{trans}[j']$.
Thus on the assumption that result holds for $t = k$, we have shown it for $t = k+1$. And so our result holds for all t, by induction. ⌟

2.3 Stable Models

In Chapter 3 we will bring together logic programming and $\mathcal{C}+$, finding ways of casting $\mathcal{C}+$ action descriptions into a form closely related to that of the Event Calculus (see Section 2.4 for background material on the latter). In describing the nature of the relation between these two formalisms, we will need to have chosen a semantics for the logic programs, so that action descriptions and the logic programs into which we shall translate them can be precisely compared. We here give the necessary details and notation for our choice of a logic-programming semantics, the *stable model* semantics ([GL88]) for logic programs with negation. In the following, we assume some familiarity with the nature and notation of logic programming. (If required, the Gelfond and Lifschitz paper [GL88] gives a rough overview of additional details; for a more comprehensive treatment, consult [Llo87] or [Hog90].)

The motivation behind the stable model semantics was to enlarge the set of logic programs which had a uniquely defined declarative semantics, and to do so in a natural way. Logic programs are to be seen as sets (possibly infinite) of ground rules

$$A \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n \tag{2.4}$$

where A, the A_i, and the B_i are atomic, and where there need be no positive literals A_i or negative literals $\neg B_i$ at all (there may also be none of either). As usual with semantics for logic programs, we consider only Herbrand models of the language. Let M be a set of atoms, and P a set of program clauses of the form (2.4). The *reduct* of P with respect to M, written P^M, is defined to be

$$\{A \leftarrow A_1, \ldots, A_m \mid (A \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n) \in P \;\wedge\; \forall i \leqslant n (B_i \notin M)\}$$

Less formally: to reduce by M, throw away any rule with $\neg B_i$ in its body if B_i is in M; then delete all the negative literals from the rules remaining.

The reduct P^M of any logic program having rules (2.4) has no negative literals whatsoever, and so has a least Herbrand model, which we will write in traditional fashion as $T^{\omega}_{P^M}(\emptyset)$, where T_P is the 'immediate consequence operator' familiar in logic programming. If this least Herbrand model of the reduct is identical to our original set of atoms M, then M is said to be a *stable model* of the logic program P. (Programs may clearly have more than one stable model, although it is stipulated by the originators of the semantics that "the *stable model semantics* is defined for a logic program Π, if Π has exactly one stable model".)

2.4 Event Calculus

The *event calculus* was first introduced by Kowalski and Sergot in [KS86], as a way of using logic programming to represent and draw inferences about the effects of events on systems in which they occur. Partly influenced by the situation calculus of McCarthy and Hayes [MH69], the event calculus is distinct from the former in its focus on the concept of *event*. The event calculus is remarkable not least for its method of answering queries about the value of a given fluent C at a given time by only considering information which might be *relevant* to that fluent's current value. When implemented as a logic program, the background of events which perturb the value of C is considered; those events may have preconditions which are the holding of other fluent atoms, whose values must be checked according to the same procedure, by looking at the events which affect their value, and so on. This way of proceeding contrasts sharply with many other systems for temporal reasoning, where entire models of a narrative of events must be handled in all their, frequently unmanageable, complexity.

Many variants of the original axioms now exist, to incorporate treatment for incompatible fluents, multi-valued fluents, the calculation of periods for which a given fluent has a given value, and so on. Recent work on the event calculus has included the application to it of a semantics based on circumscription ([Sha95]; for circumscription see [McC80] or, more thoroughly, [Lif94]); the incorporation of a treatment for continuous change [Sha90]; and abductive uses of the event calculus for planning tasks [Sha00].

We will describe a simple, standard, logic-programmed variant of the original event calculus. This formulation has been chosen because it shows most clearly the connections to the logic-programmed form of $\mathcal{C}+$ action descriptions which we will present in Chapter 3, and indeed, directly inspired these logic programs. For an overview of other approaches to the event calculus, see the articles [Sha99], or the books [Mue06a] (on formalisms for common-sense reasoning, and reasoning about action and change specifically, but with particular reference to the event calculus) or [Sha97] (on the event calculus and its relation to the frame problem).

So, for us, an *event calculus program* has four components: (*i*) standard axioms; (*ii*) a definition of `initiates/3`, which describe the effects of actions on fluent constants; (*iii*) a definition of `initially/1`, to establish initial conditions of the system (what holds at time 0); and (*iv*) a narrative of events, facts of

`happens/2`. We examine each in turn.

We take the following as our standard axioms, *Ax*:

```
holds_at(C=V, T) :-
  0 =< T,
  initially(C=V),
  \+ broken(C=V, 0, T).

holds_at(C=V, T) :-
  happens(A=V', T1),
  T1 < T,
  initiates(A=V', C=V, T1),
  \+ broken(C=V, T1, T).

broken(C=V, T1, T) :-
  happens(A=V', T2),
  T1 =< T2,
  T2 < T,
  terminates(A=V', C=V, T2).

terminates(A=V', C=V, T) :-
  initiates(A=V', C=V1, T),
  V1 \= V.
```

Their purpose is to describe how the current value of a fluent constant depends on the relevant history of the narrative. The first clause expresses that $c{=}v$ is true at t if c had value v initially, and if that value has not been disturbed in the meantime. The second clause states that c *also* has value v when an event occurred which made it take that value, and again, nothing untoward has happened since, making c take another value. The last two axioms describe how values of fluent constants are prevented from persisting by default: this occurs when they are caused to have a value other than their current. It can be seen that these axioms are not domain-specific, and assume little about the way in which systems evolve. We rely on negation-by-failure to express the default persistence of fluents.

An atom

```
initiates(a=v', c=v, t)
```

expresses that an event $a{=}v'$ occurring at time t initiates a period of time during which c has the value v (that period is imagined to start immediately *after* t, so that at t itself c may have a different value). In the definition of `initiates/3` in an event calculus program, the T is typically a free variable (expressing a form of temporal invariance of the laws of event initiation), and the body of clauses defining this predicate is either empty or a conjunction of `holds_at/2` atoms, giving the fluent preconditions of the 'successful' (relative to the initiation of the given fluent c's value) occurrence of the event $a{=}v'$:

```
initiates(A=V', C=V, T) :-
  holds_at(C1=V1, T),
  ...
  holds_at(Cn, Vn), T).
```

We will call a clause having this form an 'effect axiom', following common usage.

The final two components of event calculus programs ought to be self-explanatory: `initially(c=v)` of course expresses that c has an initial value of v, and `happens(a=v, t)` states that event $a{=}v$ occurs at time t.

Definition 2.15 Let σ be a multi-valued Boolean signature, partitioned into fluent constants and action constants. A *simplified event calculus program* P (with signature σ) is a tuple $(Ax, E, Init, N, T)$, where

- Ax is the set of standard axioms given above;
- E is a set of 'effects axioms' of the form given above;
- *Init* is a set of atoms of the form `initially(c=v)`, for $c \in \sigma^f$ and $v \in dom(c)$;
- N is a set of atoms of the form `happens(c=v, t)`, for $c \in \sigma^a$ and t such that $0 \leqslant t < m$;
- T is a set of non-negative integers $\{0, \ldots, m\}$.

We say that P is *causally definite* (the qualification avoids confusion with 'definite' as applied to clauses) when no `c=v` occurs in the head of infinitely many effects axioms. Further, a program P is said to be *complete* when:

- for all t with $0 \leqslant t < m$ and $a \in \sigma^a$, there is an atom `happens(a=v,t)` $\in N$; and
- for all $c \in \sigma^f$, there is an atom `initially(c=v)` $\in$ *Init*.

P is said to be *consistent* when there is only one such atom `happens(a=v,t)` and `initially(c=v)` in each case. ⌟

Complete and consistent event calculus programs will be the focus of our interest: they provide full specifications of the initial state of a system, and of the narrative of events and actions performed within that system.

We will hold fixed the interpretation of the atoms `t1 < t2` and `t1 =< t2`, and only consider Herbrand models M of the language of our event calculus programs which have (`t1 =< t2`) $\in M$ iff $t_1 \leqslant t_2$, and (`t1 < t2`) $\in M$ iff $t_1 < t_2$. We impose similar conditions of appropriateness on the interpretation of `\=`. Furthermore, when grounding the clauses of our program, we insist that the groundings pay attention to the intended meaning of the clauses, so that the variables `T` of the clauses in Ax, for instance, are never grounded as anything other than terms which 'represent' integers (`0`, `1`, etc.): the same goes for variables which we wish to stand for fluent constants, values of action atoms, and so on. Herbrand models of our event calculus which satisfy all of these constraints are deemed *acceptable*.

In general, stable models of our event calculus programs P need represent only some of the information about a narrative: they are, in general, *partial narratives*. If, for example, we fail to specify information in *Init* which is sufficient uniquely to identify the initial state of our system, then our stable models will reflect this: for no fluent constant c without an atom `initially(c=v)` in `Init` can there be an atom `holds_at(c=v,0)` in a stable model, regardless of the way the program's other details are fleshed out.

The acceptable stable models of a complete, consistent (in the senses defined above) event calculus program will correspond to runs through the transition system defined by an equivalent $\mathcal{C}+$ action description D.

2.5 Model Checking

The model-checking problem is that of verifying whether the behaviour of a system conforms to specification. The systems whose behaviour is being verified are modelled as finite state machines or Kripke structures; most often the user works with formal languages which define finite state machines, and from which the finite state machines may be automatically constructed. The properties one wishes to verify (frequently, liveness and safety constraints) are expressed in a temporal logic such as a variant of LTL [Pnu81] or CTL [BAPM83, CE81], and then the model-checker attempts to find a run through the system which satisfies the negation of the specification. If no such run exists, then the system satisfies the specification. If a run is found, this is a counterexample to the specification, and the model-checker outputs the undesired run, demonstrating that the system fails to work as intended, and also showing how the undesirable behaviour arises—thus giving indications how the behaviour of the system might be changed to be in conformity with specification. For an overview of model-checking consult [CGP99] or [CS01]. We will not give details of model-checking using BDDs (one of the two methods used most commonly), but will present a brief overview of *bounded model checking*, as it is more closely related to our work in Chapter 5 of this thesis. The overview presupposes some acquaintance with LTL.

2.5.1 Bounded Model Checking

We here present a brief introduction to bounded model-checking (BMC). The material in this section is standard.

Model-checkers initially relied on binary decision diagrams for their underlying representations, which were eventually succeeded by reduced, ordered, binary decision diagrams. This technology has, as is well known, been very effective in greatly expanding the cardinality of the state-space of systems with which model-checkers can cope, to the point where many industrial manufacturers routinely incorporate model-checking technology into the quality assurance phase of their production process. In the last decade, however, *bounded model checking* (first presented in [BCCZ99]) has been developed as an alternative approach: a method where runs of increasing length are successively considered as possible counterexamples to the specifications, and where a propositional formula true if such a counterexample exists is constructed and passed to an external satisfaction solver. The motivation for this alternative approach has been the enormous improvements over the recent decade or so in SAT-solving technologies. There is little correlation between problems which the two different technologies treat efficiently, and bounded model-checking (BMC) often proves to be more efficient on systems with small state-spaces. We follow [BCC$^+$03] for terminology in the succeeding concise presentation of bounded model checking.

Let the Kripke structure representing the behaviour of the system which is to be checked be $M = (S, I, T, L)$, where

- S is the set of *states* of the system;
- $I \subseteq S$ is the set of possible *initial states*—we will let $I(s)$ denote $s \in I$;
- $T \subseteq S \times S$ is the transition relation between states; and
- L is an evaluation function, $L : S \to \wp(A)$, where A is a set of atomic propositions.

Clearly, a sequence $(s_0, \ldots, s_k)$ is a path through this structure if and only if it satisfies the formula

$$[\![M]\!]_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}).$$

Specifications are expressed in LTL; in the following, $p \in A$.

$$F ::= p \mid \neg F \mid F_1 \wedge F_2 \mid \mathbf{X}F \mid \mathbf{G}F \mid \mathbf{F}F \mid F_1 \mathbf{U} F_2 \mid F_1 \mathbf{R} F_2$$

Sentences of LTL are evaluated with respect to infinite-length paths through the Kripke-structure M. Let $\pi = (s_0, s_1, \ldots)$ be such a path, let $\pi(n)$ be the n^{th} state on the path, and let π_n be the infinite-length subsequence $(s_n, s_{n+1}, \ldots)$. Then:

$$\begin{array}{lll}
\pi \models p & \text{iff} & p \in L(\pi(0)) \\
\pi \models \neg F & \text{iff} & \pi \not\models F \\
\pi \models F_1 \wedge F_2 & \text{iff} & \pi \models F_1 \text{ and } \pi \models F_2 \\
\pi \models \mathbf{X}F & \text{iff} & \pi_1 \models F \\
\pi \models \mathbf{G}F & \text{iff} & \pi_i \models F, \text{ for all } i \geqslant 0 \\
\pi \models \mathbf{F}F & \text{iff} & \pi_i \models F \text{ for some } i \geqslant 0 \\
\pi \models F_1 \mathbf{U} F_2 & \text{iff} & \pi_i \models F_2 \text{ for some } i \geqslant 0 \\
 & & \quad \text{and } \pi_j \models F_1 \text{ for all } j \text{ with } 0 \leqslant j < i
\end{array}$$

These definitions are all standard.

With bounded model checking we specify a bound k, and consider the initial fragment $(s_0, \ldots, s_k)$ of infinite-length paths through the Kripke structure which models our system. A specification g which we want to be true of the system is negated (let f be the negated specification, $f = \neg g$), and the substantial part of the model-checking process is to find a run $(s_0, \ldots, s_k)$ which makes f true. If such a run is found, then the system fails to conform to the specification g—it is easy to show that where a run of given length makes a formula f of temporal logic true, then any run of a longer length must do so too: and so must infinite-length runs of length ω. On the other hand, if there is no $(k+1)$-length run $(s_0, \ldots, s_k)$ which makes f true, we are faced with two possibilities: either this is an accurate reflection of the system, in the sense that no run of any length would invalidate g; or else, we have simply set the bound k too low, and if we were to raise it, a longer run would be found which made f true and showed our specification to be unfulfilled. Given the presence of these two possibilities, bounded model checking proceeds by gradually increasing k, until (i) f is shown to be true of a sufficiently long run, (and so g, our specification, is false) (ii)

the process becomes intractable, or (*iii*) we reach a value of k which has been shown, by analytic methods, to be large enough that if f is not found to be true of any run of a run that length, no longer run will make f true either. As one might expect, much work is accordingly directed towards finding values of k which can be proved to be adequately high.

It is important to note that initial sequences $(s_0, \ldots, s_k)$ of infinite-length runs may, though containing only $k+1$ states, represent an infinite path if there is a loop back from the final state to any other state in the sequence. Accordingly, the translation of our negated specification f into a propositional formula will have two disjoined parts, one covering the case where there is such a loop, and the other for the case where no such loop exists. This disjunction will be conjoined to $[\![M]\!]_k$ and then sent for solution to the SAT-solver.

There is a loop from s_k to s_l $(0 \leqslant l \leqslant k)$ when $T(s_k, s_l)$. Let L_k be shorthand for the formula

$$\bigvee_{l=0}^{k} T(s_k, s_l)$$

known as the *loop condition*, which represents that there is a loop back from the final state s_k of a $(k+1)$-length run to an earlier state. The translation of the negated specification f to propositional form, for passing to the SAT-solver, depends on whether the loop condition holds.

First, the case when there is a loop; to start, we need some additional notation. Where there is a loop from s_k to s_l, we set:

$$succ(i) = \begin{cases} i+i & \text{if } i < k, \\ l & \text{otherwise.} \end{cases}$$

All negation symbols $\neg$ are first moved within subformulas so that their scope is purely atomic—the negated specification is put into *negation normal form*. With a loop, the translation of the LTL formula f is given as ${}_l[\![f]\!]^0_k$, where:

$$\begin{aligned}
{}_l[\![p]\!]^i_k &:= \mathbf{t} && (\text{if } p \in L(s_i)) \\
{}_l[\![p]\!]^i_k &:= \mathbf{f} && (\text{if } p \notin L(s_i)) \\
{}_l[\![\neg p]\!]^i_k &:= \mathbf{f} && (\text{if } p \in L(s_i)) \\
{}_l[\![\neg p]\!]^i_k &:= \mathbf{t} && (\text{if } p \notin L(s_i)) \\
{}_l[\![f_1 \vee f_2]\!]^i_k &:= {}_l[\![f_1]\!]^i_k \vee {}_l[\![f_2]\!]^i_k \\
{}_l[\![f_1 \wedge f_2]\!]^i_k &:= {}_l[\![f_1]\!]^i_k \wedge {}_l[\![f_2]\!]^i_k \\
{}_l[\![\mathbf{G} f]\!]^i_k &:= {}_l[\![f]\!]^i_k \wedge {}_l[\![\mathbf{G} f]\!]^{succ(i)}_k \\
{}_l[\![\mathbf{F} f]\!]^i_k &:= {}_l[\![f]\!]^i_k \vee {}_l[\![\mathbf{G} f]\!]^{succ(i)}_k \\
{}_l[\![f_1 \mathbf{U} f_2]\!]^i_k &:= {}_l[\![g]\!]^i_k \vee ({}_l[\![f]\!]^i_k \wedge {}_l[\![f_1 \mathbf{U} f_2]\!]^{succ(i)}_k) \\
{}_l[\![f_1 \mathbf{R} f_2]\!]^i_k &:= {}_l[\![g]\!]^i_k \wedge ({}_l[\![f]\!]^i_k \wedge {}_l[\![f_1 \mathbf{U} f_2]\!]^{succ(i)}_k) \\
{}_l[\![\mathbf{X} f]\!]^i_k &:= {}_l[\![f]\!]^{succ(i)}_k
\end{aligned}$$

This is the form of the translation for the case of a loop which is given in [BCC+03], although it has flaws—the translation to propositional form ${}_2[\![\mathbf{G} p]\!]^0_3$, for instance, would be a formula with an infinite number of conjuncts:

$$p(s_0) \wedge p(s_1) \wedge p(s_2) \wedge p(s_3) \wedge p(s_2) \wedge p(s_3) \wedge \cdots$$

Any formula f which contains a temporal operator other than $\mathbf{X}$ will, when translated for the case where there is a loop, result in an infinite conjunction. The details of the remedy for this are not hard to sketch out (they depend upon recording whether the subformula beginning with a temporal operator has already been expanded once round the loop), and this remedy is essential when it comes to encoding the translation scheme in an algorithm. However, we omit the details here: it should be clear how they would proceed.

Where there is no loop in the run from s_0 to s_k, the loop formula L_k is false, and the translation $[\![f]\!]_k^i$ of our negated specification f must be different. Where $i \leqslant k$:

$$\begin{aligned}
[\![p]\!]_k^i &:= \mathbf{t}, && (\text{if } p \in L(s_i)) \\
[\![p]\!]_k^i &:= \mathbf{f}, && (\text{if } p \notin L(s_i)) \\
[\![\neg p]\!]_k^i &:= \mathbf{f}, && (\text{if } p \in L(s_i)) \\
[\![\neg p]\!]_k^i &:= \mathbf{t}, && (\text{if } p \notin L(s_i)) \\
[\![f_1 \vee f_2]\!]_k^i &:= [\![f_1]\!]_k^i \vee [\![f_2]\!]_k^i \\
[\![f_1 \wedge f_2]\!]_k^i &:= [\![f_1]\!]_k^i \wedge [\![f_2]\!]_k^i \\
[\![\mathbf{G}f]\!]_k^i &:= [\![f]\!]_k^i \wedge [\![\mathbf{G}f]\!]_k^{i+1} \\
[\![\mathbf{F}f]\!]_k^i &:= [\![f]\!]_k^i \vee [\![\mathbf{F}f]\!]_k^{i+1} \\
[\![f_1 \mathbf{U} f_2]\!]_k^i &:= [\![f_2]\!]_k^i \vee ([\![f_1]\!]_k^i \wedge [\![f_1 \mathbf{U} f_2]\!]_k^{i+1}) \\
[\![f_1 \mathbf{R} f_2]\!]_k^i &:= [\![f_2]\!]_k^i \wedge ([\![f_1]\!]_k^i \vee [\![f_1 \mathbf{R} f_2]\!]_k^{i+1}) \\
[\![\mathbf{X}f]\!]_k^i &:= [\![f]\!]_k^{i+1}
\end{aligned}$$

The base case occurs with $i = k + 1$:

$$[\![f]\!]_k^{k+1} := \mathbf{f}.$$

Thus, we presume that all formulas which reach beyond the end of our $(k+1)$-length path are false, and it is for this reason that bounded model-checking is incomplete, in the sense described earlier. Where a model of the formula sent to the SAT-solver is found, this *does* represent a counterexample to the specification. However, if no model is found, this may simply indicate that we have set the bound k too low, and if we were to increase it, a counterexample to our specification would be discovered.

The different components are yoked together into the formula $[\![M, f]\!]_k$, defined as follows:

$$[\![M]\!]_k \wedge \left(\left(\neg L_k \wedge [\![f]\!]_k^0 \right) \vee \bigvee_{l=0}^{k} \left(T(s_k, s_l) \wedge {}_l[\![f]\!]_k^0 \right) \right)$$

It is this formula which is sent to the propositional SAT-solver. To recap:

- $[\![M]\!]_k$ encodes the transition system;
- f is the negation of the specification we wish to prove our system fulfils;
- $\left(\neg L_k \wedge [\![f]\!]_k^0 \right)$ covers the case where there is no loop from s_k to a state in the run;
- $\bigvee_{l=0}^{k} \left(T(s_k, s_l) \wedge {}_l[\![f]\!]_k^0 \right)$ covers the case where there is a loop back from s_k to somewhere in the run.

2.6 Related Work

2.6.1 Action Descriptions and Extended Logic Programs

Section 7.2 of [GLL$^+$04] explores the relationship between Boolean causal theories and extended logic programs. Since, under parameterization by a non-negative integer m, an action description D of $\mathcal{C}+$ corresponds to a causal theory Γ_m^D (see Section 2.1.4), the work of [GLL$^+$04] shows how to relate $\mathcal{C}+$ to (extended) logic programs.

Let Γ be a Boolean causal theory, whose rules have the form

$$l_0 \Leftarrow l_1 \wedge \cdots \wedge l_n, \tag{2.5}$$

where each l_i is a literal: thus the causal theories in question may be indefinite (if this is taken to be the opposite of being definite), though they must not have $\bot$ in the head. The extended logic program corresponding to Γ, which we write as $elp(\Gamma)$ (Giunchiglia et al. in [GLL$^+$04] give no notation) is defined to be the set of clauses

$$l_0 \leftarrow not\ \overline{l_1}, \ldots, not\ \overline{l_n}$$

for each rule (2.5) in Γ. The following is the main theorem.

Theorem 2.16 Let Γ be a causal theory whose rules have the form (2.5). An interpretation I is a model of Γ (i.e., in our notation, $I \models_{\mathsf{C}} \Gamma$) iff I is an answer set for $elp(\Gamma)$.

Proof: This is Proposition 11 of [GLL$^+$04]. ⌟

The restriction to interpretations I is important, for in general, answer sets of extended logic programs need be neither complete nor consistent: that is, they need not contain a literal p or $\neg p$ for every atom p of the program's language, and sometimes they may contain both p and $\neg p$ [GL91]. And if an answer set I or a program $elp(\Gamma)$ is either incomplete or inconsistent, then I can clearly not be a model of Γ.

As a consequence of the surge of interest in answer-set programming in recent years, there are many implementations which find answer sets for extended logic programs. Accordingly, another implementation route for $\mathcal{C}+$ action descriptions based on logic-programming would be to reduce a set of causal laws D to the causal theory Γ_m^D, then send $elp(\Gamma_m^D)$ to an answer-set solver such as *DLV* or *SModels*. Queries of the standard form possible in CCalc could also easily be translated. Of course, CCalc itself could be used for the preliminary stage of translation to Γ_m^D: it already performs this manipulation.

2.6.2 Dependence and Acyclicity

The concept of dependence which we will define in Section 3.1 is related to that of an *acyclic action description*, as presented in [FL05], which we now describe. Let D be an action description. Where s is a state of an action description D, and α is an interpretation of σ^a which is *executable*[3] in that state, the authors

[3]The definition of when an action α is executable in a state s is not what one might expect. Intuitively, α is executable in s when there is at least one s' such that (s, α, s') is an edge of the labelled transition system defined by D; but the authors define an action α to be executable when there is no law **nonexecutable** F **if** G in D such that $s \cup \alpha \models F \wedge G$. Even given that there is no such **nonexecutable** law, it still may be true that there is no such s': in the authors' terminology, we could have that α is executable in s, but $\Phi(s, \alpha)$ empty.

define $T_{s\cup\alpha}$ to be the set of all $F \Leftarrow G$ such that

- there is a static law **caused** F **if** G in D; or
- $s \cup \alpha \models G$, for some law **caused** F **if** G **after** G in D.

(The authors of [FL05] are working with an earlier version of $\mathcal{C}+$ which does not contain action dynamic causal laws.) A causal theory Γ is said to be *acyclic relative to* $C \subseteq \sigma$ iff (i) the head of every causal rule in Γ is atomic or the negation of an atom, and (ii) there exists a mapping κ from C to the non-negative integers such that $\kappa(c_1) < \kappa(c_2)$ for all $c_1, c_2 \in C$ such that c_1 is in the head, and c_2 the body, of a rule in Γ. This definition gets lifted to action descriptions of $\mathcal{C}+$ in the following way. Let D be an action description. D is *acyclic* iff

- the set of all rules $F \Leftarrow G$, for static rules **caused** F **if** G in D, is acyclic relative to the statically-determined constants[4] of the signature σ of D; and
- for each state s and action α executable in s, $T_{s\cup\alpha}$ is acyclic with respect to σ.

The relevance to the work of [FL05] is that where an action description is acyclic, then given an interpretation of the simple fluent constants σ^{smpl} of the signature, one may calculate in polynomial time whether this interpretation can be extended to a state s.

In Section 3.1.1 we will remark on the relations between this concept of acyclicity and the notion of dependence to be defined in Section 3.1.

2.6.3 The Language $\mathcal{E}$

The language $\mathcal{E}$ was first introduced in [KM97b], and like $\mathcal{C}+$ is a high-level action language for reasoning about the effects of actions on systems over time. It also has a close relationship to the event calculus, as well as an implementation as a logic program (see [KMT01]).

The language takes inspiration from two streams of work on reasoning about time and change within artificial intelligence. On the one hand, it is related to action languages of the lineage to which $\mathcal{C}+$ belongs: the language $\mathcal{A}$, an ancestor of $\mathcal{C}+$, is cited as a direct precursor. From this research, the language $\mathcal{E}$ takes the model of a high-level formal language, with its own syntax and semantics, which is easily understood and used for reasoning about action and change, and which may be compiled down into lower-level formalisms for the purposes of implementation or problem-solving.

Unlike the language $\mathcal{A}$, however, $\mathcal{E}$ makes time ontologically primary, and this differs from approaches to temporal reasoning in the tradition stemming from the situation calculus [MH69]. With $\mathcal{E}$, time is conceived as an ordered set of points, where the ordering may be branching or linear, and events are then associated with different points in the preexistent, and in this way independent, structure. As the authors of $\mathcal{E}$ indicate, this is more convenient for

[4] The authors of [FL05] also include the rigid constants of the action description here, but they are working with an outmoded version of $\mathcal{C}+$, in which rigidity is a matter of the signature rather than laws.

certain types of reasoning about narratives: for example, the insertion of observations into a narrative is much more easily accomplished if one can simply assert '*waving* **holds-at** 4', and not have to refer to, say, the situation

$$\textit{Result(Wait, Result(Shoot, Result(Wait, Result(Load, } S_0\textit{))))},$$

as might be necessary in the situation calculus.

We will present a version of the language $\mathcal{E}$ which includes state constraints—they are called *r-propositions* by the authors [KM97a].

A *domain language* of $\mathcal{E}$ is a 4-tuple $(\Pi, \preccurlyeq, \Delta, \Phi)$, where $\preccurlyeq$ is a partial ordering over the set of *time points* Π. Δ are the *action constants* and Φ, the fluent constants; literals are constants or their negations. *Domain descriptions*, the equivalent of action descriptions in $\mathcal{C}+$, are tuples $(\gamma, \eta, \tau, \rho)$ containing four different types of proposition.

- γ is a set of *c-propositions*, of the form

$$A \textbf{ initiates } F \textbf{ when } C \tag{2.6}$$

$$\text{or} \quad A \textbf{ terminates } F \textbf{ when } C \tag{2.7}$$

  where $F \in \Phi$, $A \in \Delta$, and C is a set of fluent literals;

- η is a set of *h-propositions*

$$F \textbf{ happens-at } T, \tag{2.8}$$

  where $A \in \Delta$ and T is a time-point;

- τ is a set of *t-propositions*, having the form

$$F \textbf{ holds-at } T, \tag{2.9}$$

  with L a fluent literal and $T \in \Pi$; and

- ρ are the *r-propositions*, of the form

$$L \textbf{ whenever } C \tag{2.10}$$

  for L a fluent literal and C a set of fluent literals.

The significance of the different kinds of proposition should be obvious to anybody familiar with the event calculus, from which the terminology has been borrowed. We do not go further into the details of the semantics here, but refer the reader to the paper [KM97a], where definitions and examples will be found.

However, we will remark that the treatment of a *model* is rather different to that of the causal theories which underlie $\mathcal{C}+$. With $\mathcal{C}+$, one typically supplies a parameter m which marks the length of run to be considered, and also a number of causal laws

$$(c_1[t_1]{=}v_1) \Leftarrow \top, \ldots, (c_n[t_n]{=}v_n) \Leftarrow \top,$$

where each $c_i[t_i]$, $0 \leqslant i \leqslant n$, must belong to the signature of Γ_m^D. These causal laws constitute the query, and in effect are a partial interpretation I^- of the signature of Γ_m^D: the partial interpretation which assigns v_1 to $c_1[t_1]$, and similarly all the way up to v_n and $c_n[t_n]$. Finding models of the causal theory Γ_m^D

accompanying query then means filling out this interpretation to find a member $I = I^- \cup I^*$ of $\mathrm{I}(\sigma_m)$. The important point here is that the component I^+ may introduce new actions into the narrative, so that finding a model of an action description and query is tantamount to filling in the values of fluent and action constants both. With $\mathcal{E}$, the process is different. Interpretations are mappings from $\Phi \times \Pi$ to the set of (Boolean) truth-values. Given a domain language and domain description (where the components η and τ of the domain description correspond to the query), to find a model is thus to find an interpretation of the fluent constants which fits: supplying the remaining values for fluent constants (other than those constrained by the h-propositions), but leaving the set of h-propositions unchanged. Thus what answers to the finding of a model for Γ_m^D and accompanying query, is not the finding of a model of a domain description of $\mathcal{E}$, but the process of supplementing the set η to find models of the modified domain description.

Let $(\Pi, \preccurlyeq, \Delta, \Phi)$ be a domain language, and $(\gamma, \eta, \tau, \rho)$ a domain description of $\mathcal{E}$. We will assume that the time-points Π are a set of non-negative integers $\{0, \ldots, m\}$[5] and that $\preccurlyeq$ is the natural ordering $\leqslant$. This is a substantial restriction on the form of time underlying the narrative structures possible within $\mathcal{E}$, but many of the most common domains in AI applications can be represented by narratives constrained in this way. We will say that any domain description of $\mathcal{E}$ whose domain language falls within this restricted subset is a *restricted domain description*. (The restriction makes the relationship between $\mathcal{C}+$ and $\mathcal{E}$ much easier to express.) We also insist that there are no h-propositions of the form

$$A \textbf{ happens-at } m$$

in η (m, recall, is the length of the narrative).

Definition 2.17 Let $D^r = (\gamma, \eta, \tau, \rho)$ be a restricted domain description, with domain language $(\{0, \ldots, m\}, \leqslant, \Delta, \Phi)$ The $\mathcal{C}+$ *structure answering to* D^r is the triple (D, m, Q). The component m has already been given.

- D is an action description of $\mathcal{C}+$, with Boolean signature given by $\sigma^f = \Phi$, $\sigma^a = \Delta$, and whose causal laws are

  $$\textbf{inertial } c$$

  for all $c \in \sigma^f$;

  $$\textbf{exogenous } a$$

  for all $a \in \sigma^a$;

  $$\textbf{caused } F \textbf{ if } \top \textbf{ after } A \wedge \bigwedge C$$

  for all laws of form (2.6) in γ and

  $$\textbf{caused } \neg F \textbf{ if } \top \textbf{ after } A \wedge \bigwedge C$$

  for all laws of form (2.7) in γ; and

  $$\textbf{caused } L \textbf{ if } \bigwedge C$$

  for all r-propositions (2.10) in ρ.

[5] It would be easy also to cope with the case $\Pi = \mathbb{N}$, though for the sake of simplicity in presentation, we do not do that here.

- Q is the set

$$\{F[t] \Leftarrow \top \mid \text{there is a law (2.9) or (2.8) in } \tau \text{ or } \eta\}$$ ⌟

It should be clear why the choices of full inertia and exogeneity have been made for fluent and action constants of $\mathcal{C}+$: the semantics of $\mathcal{E}$ makes every fluent constant inertial automatically, and $\mathcal{E}$ allows any action to occur at any time.

Now, models of variants of the domain description D^r (variants made by supplementing the set η, as described above, are in one-to-one correspondence with appropriate models of the $\mathcal{C}+$ structure answering to D^r.

Theorem 2.18 Let $D^r = (\gamma, \eta, \tau, \rho)$ be a restricted domain description of $\mathcal{E}$. Let η^* be the set of h-propositions A **happens-at** T $(0 \leqslant T \leqslant m-1)$ which are not members of η. Then for all $\eta^+ \subseteq \eta^*$, the function

$$\begin{aligned} f : \mathit{models}((\gamma, \eta \cup \eta^+, \tau, \rho)) \\ \rightarrow \mathit{models}(\Gamma_m^D \cup Q \cup \{A[t] \Leftarrow \top \mid (A \textbf{ happens-at } T) \in \eta^+\}) \end{aligned}$$

is bijective, where $f(H)$, for all $H \in \mathit{models}((\gamma, \eta \cup \eta^+, \tau, \rho))$, is given by:

$$f(H)(c[t]) = \begin{cases} H(c,t), & \text{for } c \in \sigma^f \text{ and } t \text{ s.t. } 0 \leqslant t \leqslant m \\ \mathbf{t}, & \text{for } c \in \sigma^a, \text{ and if } (A \textbf{ happens-at } T) \in \eta \cup \eta^+ \\ \mathbf{f}, & \text{otherwise (for all other } c \in \sigma^a). \end{cases}$$

Proof: By induction on the length m of narrative. ⌟

As has been noted, in $\mathcal{E}$ each fluent constant is inertial, something which the semantics enforces; this is reflected in the correspondence to $\mathcal{C}+$ action descriptions, which must contain laws

$$\textbf{inertial } c$$

for every member of σ^f. Now, one clear strength of $\mathcal{C}+$ is that it allows a much more nuanced treatment of inertia. This is an immediate consequence of the greater control which $\mathcal{C}+$ allows over all forms of default, inertia being one instance—that of default persistence over time—of defaults in general. In $\mathcal{C}+$ one can easily specify that only a proper subset of the fluent constants are inertial, or that, for a given fluent constant c, only certain *values* persist by default. It is also easy to express that fluents are inertial only when certain other conditions are satisfied, by including laws

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

in an action description. This nuanced treatment of inertia is also possible within $\mathcal{EC}+$, although some interactions must be ruled out (see Section 3.1 for details). We will also see that $\mathcal{EC}+$ permits a limited use of (static) defaults: a fluent constant can be declared to have a given value by default, as long as that constant has no other value by default, nor no other value which persists by default.

2.6.4 Comparative Studies

As has already been said, Chapter 3 of this thesis will explore one way in which $\mathcal{C}+$ action descriptions can be related to a logic program which is inspired by the event calculus; a theorem relating the two formalisms forms the backbone of that chapter, and a later section will go on to examine more closely the relationship between $\mathcal{C}+$, the logic programs we will present, and one variant of the event calculus—that given already in Section 2.4.

Thus, one way of looking at that part of the thesis is to see it as a strand of a much larger work of comparison between different approaches to reasoning about action and change. In a recent paper [Mue06b], Erik Mueller gives a brief account of recent comparative work of this sort: between the event calculus, the action language $\mathcal{A}$ [GL93], the language $\mathcal{E}$ of which we gave an account in Section 2.6.3, the situation calculus, and several other approaches. This sort of comparative work is often done piecemeal: each time a new formalism is presented, or an extension or variant of an old approach proposed, it is normal to describe how the new member relates to others in the group. This is how we have proceeded in the current thesis, defining several of the relationships which exist between our approach and others in the field. This ought not, however, to be taken to imply that we do not see the value of a much more systematic and rigorous approach, one which would try to situate action languages within a common framework and thus afford the possibility of a more perspicuous representation of their differences and similarities. (One step in such a direction was attempted in [BG04].) For now, however, we simply refer to some of the most relevant comparative work, that treating of formalisms closest to those we employ.

Miller and Shanahan [MS02] prove equivalence between domain descriptions of the language $\mathcal{E}$ and axioms for a common version of the event calculus, expressed in classical logic and with a circumscriptive semantics.

Erik Mueller, in the paper [Mue06b] already cited, compares several formalizations of the event calculus in classical logic, similar to those presented by Miller and Shanahan, and again with a semantics using circumscription, to a family of Temporal Action Logics (TALs; see, for example, [DGKK98]). He proves that if the two formalisms are restricted to time-structures isomorphic to $\mathbb{N}$, to fluents over which the law of inertia always holds sway, and if a number of other restrictions are made, then the event calculus and TALs are not logically equivalent. He also shows that if a further restriction is made, that of constraining actions to be what he calls "single-step", then the two formalisms *can* be shown to be logically equivalent.

Chapter 3

Efficient Computation of Narratives

As noted above in Section 2.1.9, the current software for working with $\mathcal{C}+$ becomes unusable when working with large domains. The reason is clear: it is because when answering any query, even of a single atom, CCALC must calculate what holds true at every state and over every transition of the entire run through the labelled transition system defined by an action description. Yet the action descriptions we write are frequently inherently modular (something which [LR06] seeks to exploit). It would therefore be promising, and also highly desirable, to investigate whether an alternative way of calculating answers to queries could be found, which depended only on considering information strictly relevant to the truth or falsity of formulas.

This is what we have done. The Event Calculus, to which we gave a brief introduction in Section 2.4, is an example of a system for temporal reasoning which embodies the possibility of precisely the kind of optimized approach we would like, where laws are written to express when fluents have values, and where in answering queries only the relevant laws and fluents may be considered. When answering queries using variants of the Event Calculus similar to that presented in Section 2.4, one may of course use a 'bottom-up' style of computation, which starts from information about the initial state and gradually works through the axioms, finding all information about a given narrative. One might also find stable models of an Event Calculus logic program using a system such as `smodels`;[1] again, this will produce complete information about fluents at all times of the narrative. Perhaps it is more usual, however, when working with the Event Calculus, to query the value of a given fluent at a given time using a 'top-down' approach, which looks towards the aspects of the narrative which may be relevant to determining the value of that fluent. Our question is whether we can we use a style of computation similar to that of the 'top-down' style of the Event Calculus to work with $\mathcal{C}+$.

It turns out that the answer is yes, if we make certain restrictions to $\mathcal{C}+$—fundamentally, to remove non-determinism in our action descriptions. As well as the theoretical proof of our ideas which forms the meat of this chapter, we have written an implementation of the ideas in PROLOG, a practical proof of

[1] See `http://www.tcs.hut.fi/Software/smodels/`

concept with highly encouraging results for quite complex domains.

We first describe the subset of $\mathcal{C}+$ we work with, called $\mathcal{EC}+$. After some remarks on the notion of *dependence*, we give an overview of the logic programs which embody $\mathcal{EC}+$, and how signatures and action descriptions are represented. There follows a proof that our logic programs are correct with respect to paths through the transition systems defined by $\mathcal{EC}+$ action descriptions. We describe the process of checking that the information about initial states and narratives of actions given to our implementation is consistent, and also give details of the kinds of query which our implementation provides. An example is given, and additional details about our logic programs which present the kinds of method we use for avoiding recomputation in our causal reasoning. A further example involved the Zoo World, a standard, benchmark domain for reasoning about action and change. We conclude by relating our logic programs to one variant of the Event Calculus.

3.1 Restrictions to the Language

The language with which our engine computes is a subset of $\mathcal{C}+$, called $\mathcal{EC}+$; we now proceed to define that language. The syntax alone is restricted—semantics are defined identically to those for $\mathcal{C}+$. Our signatures will be the same as before—that is to say, multi-valued and propositional. We insist that $\sigma^{stat} = \emptyset$, so that there are no statically determined fluent constants. Later versions of $\mathcal{EC}+$ may remove this limitation.

Action descriptions are again composed of static and dynamic laws, and must be 'definite' in the sense defined in Section 2.1.1. It will be recalled that static laws take the form

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

We insist that the body G should be a conjunction of fluent atoms or $\top$ (which does not reduce the expressivity of the language, merely marking a canonicity of form). Furthermore, if the same fluent atom $c{=}v$ appears both in the head and the body of a static law, then the body must consist solely in that fluent atom. In other words, we allow static laws

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

which are usually shortened to **default** $c{=}v$, but we disallow laws

$$c{=}v \textbf{ if } c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n$$

where one of the $c_1{=}v_1$ is identical to $c{=}v$ and $n > 1$. Further constraints on the presence of default laws are treated below.

Fluent dynamic laws have the form

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

In line with our treatment of static laws, we insist that the component H be a conjunction of atoms; again, this latter insistence does not reduce the expressivity of our language. Given that H is restricted to be a conjunction of atoms—which may contain either fluent or action constants—it is evident that we can write our dynamic laws in the form

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

where H is a conjunction of fluent atoms and A is a conjunction of action atoms. We frequently make use of this opportunity. The component G of our dynamic laws must always be $\top$, with the exception of the rule's taking the form (for some fluent atom $c{=}v$)

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

which is usually abbreviated to **inertial** $c{=}v$.

The only action dynamic laws whose presence we permit in action descriptions of $\mathcal{EC}+$ are those of the form

$$\textbf{exogenous } a$$

where $a \in \sigma^a$. In fact, we *insist* that a causal law of that form should occur, for every action constant a.

Action descriptions of $\mathcal{EC}+$ are sets of the restricted static and dynamic laws described above, with two additional constraints. The first was alluded to above, and in fact concerns the relationship between expressions of defaultness and inertia. Suppose an action description contains a law

$$\textbf{default } c{=}v$$

then that description must contain no law

$$\textbf{default } c{=}v' \qquad \text{or} \qquad \textbf{inertial } c{=}v'$$

for $v \neq v'$.

The second further restriction is more complex. We say that a fluent constant c *depends* on another fluent constant c' in the action description Γ if either

- there is a static law
$$c{=}v \textbf{ if } G$$
in Γ where c' occurs in G, or
- there is some fluent constant c'' and static law
$$c{=}v \textbf{ if } G$$
in Γ, such that c'' appears in G and c'' depends on c'

Our second restriction can now be expressed by saying that, except in the case of laws **default** $c{=}v$, no fluent constant shall depend on itself.

What is the rationale of these seemingly complex and arbitrary restrictions? Some of the them have been imposed in order to rule out non-determinism in our action descriptions. In a non-deterministic domain, a complete and consistent specification of the initial state of the system and complete, consistent information about the narrative of actions may not be sufficient to make the interpretation of fluent constants unique: and in querying the value of a given fluent, we wish to be able to receive a definite answer. As an illustration of how determinism can break down when defaults and statements of inertia interfere with one another in the ways we have proscribed, consider the Boolean action description shown in Figure 3.1. When the system is in the state $\{\neg p\}$ shown

$\sigma^f = \{p\},\ \sigma^a = \{a\}$

inertial p
exogenous a
default p=**t**

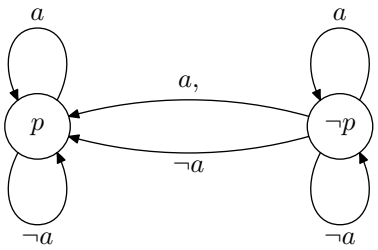

Figure 3.1: Interactions between defaults and inertia

on the right of the diagram, then any action is non-deterministic: it is clear that this is a consequence of the joint presence of

inertial p and **default** p=**t**

in the action description. For when the causal theory Γ_1^D is formed, this will have the rules

$$\neg p[1] \Leftarrow \neg p[1] \wedge \neg p[0],$$
$$p[1] \Leftarrow p[1],$$

so that (in the presence of the rest of the laws) any transition beginning in the state $\{\neg p\}$ can have a cause to be true.

The logic programs we have written, and the restrictions which have been made to $\mathcal{C}+$ in moving to $\mathcal{EC}+$, also mean that we can use SLDNF to answer queries, rather than attempting to pass the programs to an answer-set solver in order to find the entire stable model—something which would entirely defeat our purpose in the current work. This is the reason that self-dependency (apart from default laws) has been eliminated from action descriptions: for if there were a dependency loop amongst static laws, our programs would clearly cycle indefinitely. (It is a felicitous coincidence that the removal of this self-dependency also makes the proof of Theorem 3.10 much easier.)

In some circumstances, the restrictions we have imposed above on the form of action descriptions of $\mathcal{EC}+$ may somewhat be relaxed, provided (amongst other things) that the relaxation does not introduce non-determinism into the transition system. We do not comment further on the possible relaxation here, but leave it for further work.

3.1.1 Excursus on Dependence

As it will prove useful later, we now prove some theorems related to the notion of dependence. This is similar to the notion of an *acyclic* action description, defined in [FL05] and recounted by us here in Section 2.6.2. Our definition, however, is more general: there are action descriptions free from constants which depend on themselves which are not acyclic.

One reason for introducing this concept of dependence is that action descriptions in which, with the exception of **default** laws, no fluent constant depends on itself, and in which there is no interaction between **default** and **inertial** laws of the sort we proscribed in Section 3.1, are deterministic: if

$$s[0] \cup e[0] \cup s_1[1] \models_{\mathsf{C}} \Gamma_1^D \qquad \text{and} \qquad s[0] \cup e[0] \cup s_2[1] \models_{\mathsf{C}} \Gamma_1^D$$

for some action description D which conforms to our specification, then $s_1 = s_2$. This means that if we specify an initial state s_0 of a run, and also a series $e_0, e_1, \ldots, e_{t-1}$ of interpretations of σ^a, then this information can be filled out to at most one interpretation of σ_t which is a (causal-theoretic) model of Γ_t^D; and this means that if we can show a model of this causal theory does exist, then a query of some particular value of a fluent at a given time must have a definite answer—and cannot depend on which of two possible, non-deterministic courses the system takes.

The proof of Theorem 3.10, the main result of this Chapter, also relies on the series of sets we shall introduce in this section, with the aid of the concept of dependence.

Thus, let D be an action description of $\mathcal{EC}+$, and let $L \subseteq D$ be a subset of its static laws which contains no default laws, of the form $c{=}v$ **if** $c{=}v$. Define

$$C_L = \{c' \mid \exists(c{=}v \textbf{ if } c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n \in L) \wedge (c' = c \vee c' = c_1 \vee \cdots \vee c' = c_n)\}$$

so that C_L in fact is the set of all fluent constants mentioned in the static laws of L. Now, we will define a sequence $E_0, \ldots, E_j$ of subsets of C_L, such that the members of E_i do not depend on any E_k for $k \leqslant i$. This sequence will be valuable later, when we prove the correctness of our logic programs for $\mathcal{EC}+$. In order to define the sequence $E_0, \ldots, E_j$, we make use of an operator D on subsets of constants in C_L.

So first, for a set L of causal laws as before and any $S \subseteq C_L$, we define

$$D(S) = \bigcup_{c \in S} \{c' \mid \exists(c{=}v \textbf{ if } c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n) \in L \text{ and } (c' = c_1 \vee \cdots \vee c' = c_n)\}$$

It is obvious that $D(S)$ is the set of constants that members of S depend on, relative to the background set of laws L.

Theorem 3.1 Let $D^0(S) = S$, and $D^{k+1}(S) = D(D^k(S))$. We have

$$D^{k+1}(S) = \bigcup_{c \in D^k(S)} D(\{c\})$$

Proof: It is obvious from the definition of $D(S)$ that we have

$$D(S) = \bigcup_{c \in S} D(\{c\}).$$

So, substituting $D^k(S)$ for S here, we get our result. ⌟

Theorem 3.2 We have:

$$D(\{c_1, \ldots, c_n\}) = D(\{c_1\}) \cup \cdots \cup D(\{c_n\}).$$

Proof: Immediate from the definition of $D(S)$. ⌟

Theorem 3.3 The following holds.

$$D^k(S) = \bigcup_{c \in S} D^k(\{c\})$$

Proof: The proof is by induction on k.
Base (k=0): This is simply $S = \bigcup_{c \in S} \{c\}$.
Induction: Assume the result holds for $k = j$, so that we have

$$D^j(S) = \bigcup_{c \in S} D^j(\{c\}).$$

Now we must show the result for $k = j + 1$. So, as our sets S are finite (because action descriptions of $\mathcal{EC}+$ must be definite), we let $S = \{c_1, \ldots, c_n\}$, and so:

$$\begin{aligned} D^{j+1}(S) &= D(D^j(S)) \\ &= D(\bigcup_{c \in S} D^j(\{c\})) \qquad \text{(induct. hyp.)} \\ &= D(D^j(\{c_1\}) \cup \cdots \cup D^j(\{c_n\})) \end{aligned}$$

Now, letting $D^j(\{c_i\}) = \{c_i^1, \ldots, c_i^{m_i}\}$ (as these sets must be finite too), we continue:

$$\begin{aligned} D^{j+1}(S) &= D(\{c_1^1, \ldots, c_1^{m_1}, c_2^1, \ldots, c_n^1, \ldots, c_n^{m_n}\}) \\ &= D(\{c_1^1\}) \cup \cdots \cup D(\{c_1^{m_1}\}) \cup \cdots \cup D(\{c_n^{m_n}\}) \\ &\qquad \text{(by Theorem 3.2)} \\ &= \bigcup_{c \in D^j(\{c_1\})} D(\{c\}) \cup \cdots \cup \bigcup_{c \in D^j(\{c_n\})} D(\{c\}) \\ &= \bigcup_{c \in S} \bigcup_{c' \in D^j(\{c\})} D(\{c'\}) \\ &= \bigcup_{c \in S} D^{j+1}(\{c\}) \qquad \text{(by Theorem 3.1)} \end{aligned}$$

Thus, on the assumption that the result is true for $k = j$, it is shown for $k = j+1$. Therefore, by induction, we have our result. ⌟

Now, consider as before an action description D of $\mathcal{EC}+$, and let $L \subseteq D$ be a subset of the static laws of D with no **default** conditions in.

Observation 3.4 Let $S \subseteq C_L$. The set of constants on which constants in S depend and which are in C_L is given by

$$\bigcup_{i=1}^{\infty} D^i(S)$$

⌟

Theorem 3.5 For all $c \in C_L$, there is $k > 0$ such that $D^k(\{c\}) = \emptyset$.

Proof: Suppose, for contradiction, there is some $c \in C_L$ with all $k > 0$ such that $D^k(\{c\}) \neq \emptyset$. We note that for all $S \subseteq C_L$, we have $D(S) \subseteq C_L$ (by the definitions of $D(S)$ and C_L), and let $|C_L| = k'$. There must then be p, q, with $0 \leqslant p, q \leqslant k'$ and $p \neq q$, with for some $c' \in C_L$, both $c' \in D^p(\{c\})$ and $c' \in D^q(\{c\})$. Yet then c' depends on itself (by the previous observation), contradicting the definition of action descriptions of $\mathcal{EC}+$. Thus we have our result. ⌟

Theorem 3.6 Let $S \subseteq C_L$. Then there is $k \geqslant 0$ such that $D^k(S) = \emptyset$.

Proof: This follows from the previous result, Theorem 3.3, and the necessary finiteness of S. ⌟

Given the previous corollary, it is clear that, for D, L and C_L as before, there must be for some j with $0 \leqslant j$, a sequence of sets $C_L, D^1(C_L), \ldots, D^j(C_L)$, such that $D^j(C_L) = \emptyset$. We use the sequence to define another, $E_0, \ldots, E_{j-1}$, so that

$$\begin{aligned} E_0 &= D^{j-1}(C_L), \\ E_1 &= D^{j-2}(C_L) - D^{j-1}(C_L), \\ &\vdots \\ E_{j-1} &= C_L - (D(C_L) \cup \cdots \cup D^{j-1}(C_L)) \end{aligned}$$

It is clear that the union of the E_i is equal to C_L, and that by the definition of E_0, its members depend on no constant in C_L. Further, the sets in the sequence E_i must be pairwise disjoint.

Theorem 3.7 Let k be such that $0 \leqslant k \leqslant j - 1$. Then, the members of E_k depend on no constant in

$$\bigcup_{i=k}^{j-1} E_i$$

Proof: Assume for contradiction that there is some member c of E_k such that c depends on a constant c' which is a member of the union of E_i given. Then for some k', with $k \leqslant k' \leqslant j - 1$, we have $c' \in E_{k'}$. But by the definition of the E_i, we have that

$$c \in D^{j-1-k}(C_L) - \bigcup_{i=j-k}^{j-1} D^i(C_L) \quad \text{and} \quad c' \in D^{j-1-k'}(C_L) - \bigcup_{i=j-k'}^{j-1} D^i(C_L)$$

so that clearly, $c \in D^{j-1-k}(C_L)$ and $c' \in D^{j-1-k'}(C_L)$. As $k \leqslant k'$, then by Observation 3.4, c' must depend on c. But then as each depends on the other, then each depends on itself, which is a contradiction of the nature of D (and thus of L). ⌟

In Section 2.6.2 we described the concept of an *acyclic action description* as it features in [FL05]. This is clearly related to the absence of dependence in action descriptions. If an action description D of $\mathcal{C}+$ is acyclic, then no constant c of the signature of D can depend on itself. For if c did depend on itself, there would be a series of static laws

$$\textbf{caused } c_1{=}v_1 \textbf{ if } G_1, \ldots, \textbf{caused } c_n{=}v_n \textbf{ if } G_n$$

where $c_1 = c$, and such that for all i with $1 < i \leqslant n$, c_i occurs in G_{i-1}, and c_1 occurs in G_n. And in that case, there could be no mapping κ as required by the definition of acyclicity (see Section 2.6.2).

In general, the converse direction does not hold: an action description D which is free from constants which depend on themselves need not be acyclic. This is because, in defining the concept of dependence, we made use of the fact that action descriptions of $\mathcal{EC}+$ satisfy various other constraints, including a restriction on the components G of fluent dynamic laws

caused F **if** G **after** H.

We do, however, have a restricted converse: that all action descriptions of $\mathcal{EC}+$ in which no constant depends on itself (i.e. in which there are no **default** laws) are acyclic.

This concludes the interlude on dependence.

3.1.2 Action Domains

Most action descriptions presented as examples in [GLL$^+$04] are familiar problem cases for knowledge representation. It is clear that many of these action descriptions of $\mathcal{C}+$ are also descriptions of $\mathcal{EC}+$: the 'Monkey and Bananas' domain, the 'Shooting Turkeys' domain, the 'Lifting Table Ends' domain, and the 'Publishing Papers' domain. As noted, we have removed the possibility of non-determinism in our action descriptions by placing constraints on the simultaneous presence of certain sorts of causal laws which have a default component, and of course this does mean that some domains which are common in artificial intelligence cannot be successfully treated by the logic programs we will introduce.

3.2 Logic Programs

We now show how action descriptions of $\mathcal{EC}+$ are to be represented as logic programs, in a way inspired by the event calculus. The following, as one might expect, is a greatly simplified version of the actual PROLOG code of our programs: the essence only has been retained, in a way which abstracts away from methods of avoiding recomputation (see Section 3.7), from components which enable the plotting of causal histories through a 'trace' facility (see Section 3.5.1), and from predicates which print the answers to queries and effect a partial evaluation of the logic programs.

3.2.1 Signatures

Suppose we have an action description D, having signature σ, For all $c \in \sigma$ (where this c is either a fluent constant or an action constant), and for all $v \in dom(c)$, we have a fact

```
domain(c, v).
```

so that the series of `domain/2` facts show the signature of the action description. The Boolean truth values are represented as PROLOG atoms `ff` and `tt`. Further, for all $c \in \sigma^f$ we have a fact

```
flu_constant(c).
```

and for all $c \in \sigma^a$ we have a fact

```
act_constant(c).
```

3.2.2 Laws

Static laws either have $\bot$ as their head, or a fluent atom. In the former case, the law

$$\bot \textbf{ if } c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n$$

is represented as a clause

```
never([c1=v1,...,cn=vn]).
```

In the latter case, a static law is an expression

$$c{=}v \textbf{ if } c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n.$$

When such laws are not default conditions $c{=}v$ **if** $c{=}v$, then we express them as clauses

```
causes(c=v, [c1=v1,...,vn=vn]).
```

If they are defaults, then they are denoted by facts

```
default(c=v).
```

Fluent dynamic laws, just like static laws, may either have $\bot$ as their head or a fluent atom. In the former case they will have the form

$$\bot \textbf{ if } \top \textbf{ after } (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m) \wedge (a_1{=}v_1' \wedge \cdots \wedge a_n{=}v_n')$$

which we represent (paying attention to the abbreviations given in Section 2.1.6) as

```
nonexecutable([a1=v1',...,an=vn'], [c1=v1,...,cm=vm]).
```

In the latter case, they may be of two kinds. In the first, they have the form

$$c{=}v \textbf{ if } \top \textbf{ after } (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m) \wedge (a_1{=}v_1' \wedge \cdots \wedge a_n{=}v_n')$$

and will be expressed by facts

```
causes(c=v, [a1=v1',...,an=vn'], [c1=v1,...,cm=vm]).
```

The second kind is that of statements of inertia; a law

$$c{=}v \textbf{ if } c{=}v \textbf{ after } c{=}v \tag{3.1}$$

is to be represented by a logic-programmed fact

```
inertial(c=v).
```

(This is different from $\mathcal{C}+$, in which a fluent constant is declared as inertial, and this statement stands for the set of laws (3.1) for *all* values $v \in dom(c)$.)

It is clear from the preceding remarks that an action description D of $\mathcal{EC}+$ will uniquely determine definitions for predicates `never/1`, `causes/2`, `default/1`, `nonexecutable/2`, `causes/3` and `inertial/1`. The set of such definitions for the description D will be called $Laws(D)$.

3.2.3 Initial States and Actions

One frequent way of reasoning with action descriptions of $\mathcal{C}+$ is the following. One starts with an action description, which as has been shown determines a transition system—the nodes of the system representing states of the domain being modelled, and the edges representing actions performed within the domain, which change the properties of states. To this action description or transition system, one adds further information describing the known properties of an *initial state* of the system in question, together with details of the actions performed at later times. One then proceeds to ask questions about what may be inferred about later states than the initial, given what is known to hold in the initial state and what is known to have happened at the times intervening.

Given the correspondence between action descriptions (and times t) and causal theories, it is clear how one might set about answering such queries. Let D be an action description with signature $\sigma = \sigma^f \cup \sigma^a$, and suppose one is interested in properties of states at times up to t. Information about an initial state can be represented by a set $Init$ of atoms of the form $c[0]{=}v$, with $c \in \sigma^f$ and $v \in dom(c)$. Information about actions occurring later can be represented by sets of atoms $Haps_0, ..., Haps_{t-1}$, where the atoms in $Haps_i$ each have the form $a[i]{=}v$, for $a \in \sigma^a$ and $v \in dom(a)$. For the purposes of our query, we can consider the models of the causal theory

$$\Gamma^D_t \cup \{F \Leftarrow \top \mid F \in Init \cup Haps_0 \cup \cdots \cup Haps_{t-1}\}$$

(Given that there are no statically-determined fluent constants in the action descriptions of $\mathcal{EC}+$, the models of this causal theory are the same as those of Γ^D_t which make the atoms of *Init* and the $Haps_i$ true.) We may wish to ask, for example, whether some fluent atom $c[i]{=}v$, for $0 \leqslant i \leqslant t$, is true in all models of this theory, or to find out whether there is more than one model.

With our logic programs, the case is much the same. If we want to impose that in the initial state a fluent c is constrained to have value v, then we represent that by a fact

```
init(c=v).
```

Information about actions is represented similarly: if at time i we wish to state that action constant a has value v, then we include a fact

```
happens(c=v, i).
```

in our logic programs. We insist that complete and consistent information be provided about the initial state—that is to say, for all fluent constants $c \in \sigma^f$, there should be exactly one $v \in dom(c)$ such that $c[0]{=}v$ is a member of $Init$. Further, complete and consistent information should be given about the performance of actions (in other words, the truth of atoms $a{=}v$ for $a \in \sigma^a$) up to the maximum time in which we are interested. Thus, for all i ($0 \leqslant i < t$), and for every $a \in \sigma^a$, there should be exactly one $v \in dom(a)$ such that, $a[i]{=}v \in Haps_i$. It may seem that this places a large burden on the user, in that where the signature of the action description has a large number of fluent and action constants, or else where the length of narrative t is large, very many atoms will need to be specified in order to make the sets *Init* and the $Haps_i$ complete. This problem does not arise: in our implementation we have

included features which enable a very concise representation of initial states and narratives of actions, which we describe in more detail in Section 3.5.

In addition, since we are supposing the presence of a non-negative integer t, which is the maximum length of histories under consideration, we include a fact

```
max_time(t).
```

3.2.4 Axioms

The most important predicate definitions in our logic programs are inspired by the axioms of that brand of the event calculus described in Section 2.4, and are now presented. Again, recall that the axioms presented here are given in a simplified form which retains the essence of the algorithm whilst abstracting away from bookkeeping, time-saving, and other devices—some of which will be described later.

```
% --------------- caused/2 ---------------

caused(C=V, 0) :-
    init(C=V).

caused(C=V, T1) :-
    0 < T1,
    max_time(MaxT),
    T1 =< MaxT,
    causes(C=V, A, H),
    T is T1 - 1,
    all_happen(A, T),
    all_caused(H, T).

caused(C=V, T) :-
    max_time(MaxT),
    T =< MaxT,
    causes(C=V, F),
    all_caused(F, T).

caused(C=V, T1) :-
    0 < T1,
    max_time(MaxT),
    T1 =< MaxT,
    inertial(C=V),
    T is T1 - 1,
    caused(C=V, T),
    \+ clipped(C=V, T, T1).

caused(C=V, T) :-
    0 < T,
    max_time(MaxT),
    T1 =< MaxT,
    default(C=V),
```

```
    \+ overridden(C=V, T).

% --------------- all_happen/2 ---------------

all_happen([], _).

all_happen([C=V|Rest], T) :-
    happens(C=V, T),
    all_happen(Rest, T).

% --------------- all_caused/2 ---------------

all_caused([], _).

all_caused([C=V|Rest], T) :-
    caused(C=V, T),
    all_caused(Rest, T).

% --------------- clipped/3 ---------------

clipped(C=V, T1, T2) :-
    max_time(MaxT),
    T2 =< MaxT,
    0 =< T1,
    domain(C, V1),
    V1 \= V,
    (
        causes(C=V1, A, H),
        all_happen(A, T1),
        all_caused(H, T1)
        ;
        causes(C=V1, F),
        all_caused(F, T2)
    ).

% --------------- overridden/2 ---------------

overridden(C=V, T) :-
    domain(C, V1),
    V1 \= V,
    caused(C=V1, T).
```

The two definitions for `all_happen/2` and `all_caused/2` should be straightforward. We here give brief commentary on the predicates `caused/2`, `clipped/3` and `overridden/2`.

The first clause of the definition for `caused/2` expresses our idea that anything which we specify to be true in the initial state is caused to hold. The second clause refers to dynamic laws, which—it will be remembered—are stored

as facts `causes(c=v, A, H)` in our logic-programmed version of action descriptions. The clause for dynamic laws is informed by the fact that if we have a causal theory Γ_t^D containing the laws

$$c[i+1]{=}v \Leftarrow H[i] \wedge A[i]$$

for $0 \leqslant i < t$, and where H is a conjunction of fluent atoms and A is a conjunction of action atoms, then if some interpretation X of the signature of Γ_t^D is such that $X \models H[i] \wedge A[i]$, then the reduct $(\Gamma_t^D)^X$ must contain $c[i+1]{=}v$. The third clause of the definition of `caused/2` relates in a similar way to static laws of the action description D.

The fourth clause in the definition of `caused/2` concerns the circumstances under which the truth of a fluent $c{=}v$ carries through inertially from the previous state; from the definition of `clipped/3`, it is clear that this happens when the fluent has been declared to be inertial, and when the constant c is not caused to have some other value, either through a dynamic law 'activated' by the transition between states, or through a static law activated by properties of the successor state. Finally, the fifth clause ensures that a constant c is caused to have its default value v if the constant has been caused to have none of its other values (in other words, if the default value has not been overridden) much as one might expect.

The definitions of the five predicates given here are collected together in the set *Axioms*.

3.2.5 The Components Together

We now collect the clauses of the previous subsections together to form a logic program.

Definition 3.8 Let D be an action description of $\mathcal{EC}+$ and t a non-negative integer. Let also $Init$ be a set of atoms of the form $c[0]{=}v$, and $Haps_i$, for $0 \leqslant i < t$ be sets of atoms of the form $c[i]{=}v$, as previously described. Then we define

$$LP(D, t, Init, \{Haps_i \mid 0 \leqslant i < t\})$$

as the union of the sets:

- $\{$`domain(c, v)` $\mid c \in \sigma$ and $v \in dom(c)\}$
- $\{$`flu_constant(c)` $\mid c \in \sigma^f\}$
- $\{$`act_constant(c)` $\mid c \in \sigma^a\}$
- $Laws(\Gamma)$
- $\{$`init(c=v)` $\mid c[0]{=}v \in Init\}$,
- $\{$`happens(a=v,i)` $\mid a[i]{=}v \in Haps_i\}$, for all $0 \leqslant i < t$,
- $\{$`max_time(t)`$\}$.
- *Axioms* ⌟

With this crucial definition behind us, we can proceed to investigate the relations between models of causal theories and stable models of our logic programs.

3.3 Proof

We will use the stable model semantics [GL88], of which we gave an account in Section 2.3 to give the meaning of our logic programs. Though this semantics has a notion of 'reduct' which is close in spirit and notation to that for causal theories, we trust that there will be no confusion between the two in the ensuing proof and discussion.

Observation 3.9 Let D be an action description of $\mathcal{EC}+$, let t be a non-negative integer, and $Init$ and $Haps_0, \ldots, Haps_{t-1}$ be sets as previously described. Then

`domain/2`	`flu_constant/1`	`act_constant/1`
`never/1`	`causes/2`	`default/1`
`nonexecutable/2`	`causes/3`	`inertial/1`
`init/1`	`happens/2`	`max_time/1`
`caused/2`	`all_happen/2`	`all_caused/2`
`clipped/3`	`overridden/2`	

are the predicates defined in $LP(D, t, Init, \{Haps_i \mid 0 \leqslant i < t\})$. ⌟

Theorem 3.10 Let D be an action description of $\mathcal{EC}+$, with signature σ, and suppose that $X \in \mathrm{I}(\sigma_t)$. Then, where $Init$ and the $Haps_i$ are as previously given, $X \models_{\mathsf{C}} \Gamma_t^D \cup \{F \Leftarrow \top \mid F \in Init \cup Haps_0 \cup \cdots \cup H_{t-1}\}$ iff there is a stable model M of $P = LP(D, t, Init, \{Haps_i \mid 0 \leqslant i < t\})$ such that

(i) $\forall c, v, i(X \models c[i]{=}v$ iff `caused(c=v,i)` $\in M)$ $\quad (c \in \sigma^f)$

(ii) $\forall a, v, i(X \models a[i]{=}v$ iff `happens(a=v,i)` $\in M)$ $\quad (a \in \sigma^a)$

(iii) $\forall L($`never(L)` $\in P$
$\rightarrow \forall i($`all_caused(L,i)` $\notin M))$

(iv) $\forall L1, L_2($`nonexecutable(L1,L2)` $\in P$
$\rightarrow \forall i($`all_caused(L2,i)` $\in M \rightarrow$ `all_happen(L1)` $\notin M))$

Proof: $(\rightarrow)$:
Assume some interpretation X of the signature of Γ_t^D, which is a model of the causal theory $\Gamma_t^D \cup \{F \Leftarrow \top \mid F \in Init \cup Haps_0 \cup \cdots \cup H_{t-1}\}$, and let $(\Gamma_t^D)^*$ abbreviate the name of that causal theory. We will describe a Herbrand model M of $P = LP(D, t, Init, \{Haps_i \mid 0 \leqslant i < t\})$ which is a stable model of P and which satisfies the four conditions enumerated. Letting HU stand for the Herbrand Universe, we include in M:

- all facts of the following predicates in P: `domain/2`, `flu_constant/1`, `act_constant/1`, `never/1`, `causes/2`, `default/1`, `nonexecutable/2`, `causes/3`, `inertial/1`, `max_time/1`
- $\{$`init(c=v)` $\mid X \models c[0]{=}v$ and $c \in \sigma^f\}$,
- $\{$`happens(a=v,i)` $\mid X \models a[i]{=}v$ and $a \in \sigma^a\}$,
- $\{$`caused(c=v,0)` $\mid X \models c[i]{=}v$ and $c \in \sigma^f\}$,

- {`all_happen([],t)` | $t \in HU$},
- {`all_caused([],t)` | $t \in HU$},
- {`all_happen(L,i)` | L is a list of pairs `a=v`, with $X \models a[i]{=}v$ and $a \in \sigma^a$},
- {`all_caused(L,i)` | L is a list of pairs `c=v`, with $X \models c[i]{=}v$ and $c \in \sigma^f$},
- {`clipped(c=v,t1,t2)` | $\exists(c{=}v' \textbf{ if } \top \textbf{ after } H \wedge A) \in D$ such that $X \models (H \wedge A)[t_1]$ and $v' \neq v$},
- {`clipped(c=v,t1,t2)` | $\exists(c{=}v' \textbf{ if } F \in D$ such that $F \neq c{=}v$ and $X \models F[t_2]$ and $v' \neq v$},
- {`overridden(c=v,i)` | $X \models c[i]{=}v'$ and $c \in \sigma^f$ and $v' \neq v$}.

We must first show that this M is indeed a stable model of P—in other words, that it is the least Herbrand model of P^M. The first step is to show that it is a model of P^M.
So, assume for contradiction that M is not a model of P^M. Then there must be some (definite) clause whose head is false according to the interpretation M, but whose body is true. Clearly such a clause cannot have as its head any of the predicates

`domain/2`	`flu_constant/1`	`act_constant/1`
`never/1`	`causes/2`	`default/1`
`nonexecutable/2`	`causes/3`	`inertial/1`
`max_time/1`		

for all facts from P of these predicates were included in M by definition, and there are only facts in the definitions of these predicates. We are left with a case analysis.
`init/1`: In this case, we must have some `init(c=v)` $\in P$. That can only come about if $c[0]{=}v \in Init$. Since $X \models_{\mathsf{C}} (\Gamma_t^D)^*$ by hypothesis, clearly $X \models F$ for all $F \in Init$, so that $X \models c[0]{=}v$. Thus by the definition of M, we have that `init(c=v)` $\in M$, a contradiction.
`happens/2`: We must have some `happens(a=v,i)` $\in P$. Then clearly $a[i]{=}v \in Haps_i$, and so similarly to the previous case, we have $X \models a[i]{=}v$. Thus `happens(a=v,i)` $\in M$, which yields the contradiction we want.
`caused/3` (axiom 1): In this case, we must have that `init(c=v)` $\in M$, and so $X \models c[0]{=}v$. But then by definition, `caused(c=v,0)` $\in M$, a contradiction.
`caused/3` (axiom 2): Here, there would have to be an integer i ($0 \leqslant i < t$), together with a dynamic rule $c{=}v$ **if** $\top$ **after** $H \wedge A$ in D, with H as some conjunction $c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n$, A as a conjunction $a_1{=}v'_1 \wedge \cdots \wedge a_m{=}v'_m$, such that `caused(cj=vj,i)` $\in M$ for all $1 \leqslant j \leqslant n$, and `happens(aj=vj',i)` $\in M$, for all $1 \leqslant j \leqslant m$. But in that case, the definition of M evidently requires that $X \models (H \wedge A)[i]$, so that $c[i]{=}v \in ((\Gamma_t^D)^*)^X$ and thus $X \models c[i]{=}v$. So `caused(c=v,i)` $\in M$, a contradiction.
`caused/3` (axiom 3): This case is straightforward, given the model of reasoning exercised above.

`caused/3` (axiom 4): Here, we evidently have, for some i with $(0 \leqslant i < t)$, `caused(c=v,i)` $\in M$. Also, we must have (letting i' notate the successor to i) `clipped(c=v,i,i')` $\notin M$. From the latter follows that there is no dynamic law $c{=}v'$ **if** $\top$ **after** G in D $(v \in dom(c), v' \neq v)$ such that $X \models G[i]$, and also that there is no static law $c{=}v'$ **if** F $(v \in dom(c), v' \neq v)$ in D such that $X \models F[i]$. As **inertial** $c{=}v$ is in D, there can be no rules expressing default behaviour of the fluent c, and so the only *causal laws* of $(\Gamma_t^D)^*$ whose heads contain the constant c—and whose bodies have not been discounted as false—have to be the correlates of the laws of inertia, in other words the laws $c[i']{=}v' \Leftarrow c[i']{=}v' \wedge c[i]{=}v'$ for all $v' \in dom(c)$. But since $X \models_{\mathsf{C}} (\Gamma_t^D)^*$ and $X \models c[i]{=}v$ (as `caused(c=v,i)` $\in M$), the only one of the causal laws whose head contains c and has time-stamp i', and whose body may be true, is $c[i']{=}v \Leftarrow c[i']{=}v \wedge c[i]{=}v$. As X is a model for $(\Gamma_t^D)^*$, we know that there must be at least one rule whose head has time-stamp i' and contains the constant c, and whose body is true in X. Thus it must be the rule we have narrowed ourselves down to, and so also $X \models c[i']{=}v$. But then `caused(c=v,i')` $\in M$, which is a contradiction.
`caused/3` (axiom 5): We must have, for i such that $0 \leqslant i \leqslant t$, no $v' \in dom(c)$ such that $v' \neq v$ and $X \models c[i]{=}v$—because `overridden(c=v,i)` $\notin M$. But then since $X \models_{\mathsf{C}} (\Gamma_t^D)^*$, it must be the case that there is *some* $v' \in dom(c)$ such that $X \models c[i]{=}v'$. This v' can only be v, so that $X \models c[i]{=}v$ and thus by the definition of M, `caused(c=v,i)` $\in M$, which is the contradiction we want.
`all_happen/2`: Clearly, since we have included all facts `all_happen([], t)`, for $t \in HU$, then for us to have a body which is true and a head which is false, then there must be a, v, i, such that `happens(a=v,i)` $\in M$, and some list *Rest* such that `all_happen(Rest, i)` $\in M$. But then, clearly `Rest` must be a list of pairs `a'=v'` such that `happens(a'=v',i)` $\in M$. But then clearly `[a=v|Rest]` is such a list, and thus by definition, the head `all_happen([a=v|Rest], i)` $\in M$, which is the contradiction we want.
`all_caused/2`: The contradiction here is derived in the same way we did it for `all_happen/2`.
`clipped/3`: Suppose the body of the clause is true; clearly, this can happen in one of two ways. In the first case, we have three atoms `causes(c=v1,a,h)`, `all_happen(a,t1)`, `all_caused(h,t1)` in M, for some t_1 such that $0 \leqslant t_1 < t$. But then by the definition of M, we must have some law $c{=}v'$ **if** $\top$ **after** $H \wedge A$ (where H is a conjunction of the atoms represented in `h`, and likewise for A and `a`), such that $X \models (H \wedge A)[t_1]$. But then clearly `clipped(c=v,t1,t2)` $\in M$, by the definition of M. In the second way the body of a clause for `clipped/3` can be true, we must have some `causes(c=v1,f)` and `all_caused(f,t2)` in M. Now by the definition of M it is clear there must be $c{=}v_1$ **if** F in D, such that $X \models F[t_2]$. But then by definition, `clipped(c=v,t1,t2)` $\in M$, so that we have (according to both cases) our desired contradiction.
`overridden/2`: Suppose the body of the clause is true. Then there is some atom `caused(c,v1,t)` $\in M$, with $v1 \neq v$. By the definition of M, we must then have that $X \models c[t]{=}v1$, and thus again clearly `overridden(c,v,t)` $\in M$ by definition, which is a contradiction.

Thus it is clear that M is a model of P^M. We now have to show that it is the least (Herbrand) model. Letting T_P denote the 'immediate consequence' operator of Kowalski and Van Emden [vEK76], we see that the result we need is that $M \subseteq T_{P^M}^{\omega}(\emptyset)$. Since we know which predicates M contains—for we have

specified them—we show the required set-theoretic inclusion using a case-based analysis. First note that the only facts of the predicates

`domain/2`	`flu_constant/1`	`act_constant/1`
`never/1`	`causes/2`	`default/1`
`nonexecutable/2`	`causes/3`	`inertial/1`
`max_time/1`		

included in M are also facts in P—and so must necessarily be in P^M and $T^{\omega}_{P^M}(\emptyset)$.

`init/1`: Let `init(c=v)` $\in M$. Then clearly, by the definition of M, we have $X \models c[0]{=}v$. But as $Init$ completely specifies the initial state (ie., for every $c' \in \sigma^f$ there is exactly one $v' \in dom(c')$ such that $c'{=}v' \Leftarrow \top \in Init$), and as $X \models_{\mathsf{C}} (\Gamma^D_t)^*$, we must have that $c[0]{=}v \in Init$, and thus by the definition of P, clearly `init(c=v)` $\in P$, and so too `init(c=v)` $\in T^{\omega}_{P^M}(\emptyset)$, as desired.

`happens/2`: Let `happens(a=v,i)` $\in M$. Then by definition, $X \models a[i]{=}v$. As for all $a' \in \sigma^a$, there is $v' \in dom(a')$ such that $a'[i]{=}v' \in Haps_i$, and as $X \models_{\mathsf{C}} (\Gamma^D_t)^*$, we must have that $a[i]{=}v \in Haps_i$, so that by the definition of P, `happens(a=v,i)` $\in P$, and so too `happens(a=v,i)` $\in T^{\omega}_{P^M}(\emptyset)$, as desired.

`caused/2`: We show the result for atoms `caused(c=v,i)` by induction on i.

So, first consider the case for $i = 0$, and let `caused(c=v,0)` $\in M$. Then clearly, $X \models [0]c{=}v$, so that by the same reasoning for the case of `init(c=v)`, we have that `init(c=v)` $\in P^M$. But then it is clear that `init(c=v)` $\in T_{P^M}(\emptyset)$, so that given the first clause defining the predicate `caused/2`, we have `caused(c=v,0)` $\in T^2_{P^M}(\emptyset)$, and thus also `caused(c=v,0)` $\in T^{\omega}_{P^M}(\emptyset)$—which is what we want.

Now assume the result for $i \leqslant k$, and let `caused(c=v,k+1)` $\in M$. Then $X \models c[k+1]{=}v$, and we know that X is the unique model of $((\Gamma^D_t)^*)^X$ (for that is what defines $X \models_{\mathsf{C}} (\Gamma^D_t)^*$). Thus there is some law $c[k+1]{=}v \Leftarrow F \in (\Gamma^D_t)^*$, with $X \models F$, and no rule $c[k+1]{=}v' \Leftarrow G \in (\Gamma^D_t)^*$ for $v' \neq v$ and such that $X \models G$. The law $c[k+1]{=}v \Leftarrow F$ can clearly take one of four forms:

(i) $c[k+1]{=}v \Leftarrow (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m)[k] \wedge (a_1{=}v'_1 \wedge \cdots \wedge a_n{=}v'_n)[k]$

(ii) $c[k+1]{=}v \Leftarrow c[k+1]{=}v \wedge c[k]{=}v$

(iii) $c[k+1]{=}v \Leftarrow c[k+1]{=}v$

(iv) $c[k+1]{=}v \Leftarrow (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[k+1]$

Now, suppose we have a law of the form (i). We have that

$$X \models (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m)[k] \quad \text{and} \quad X \models (a_1{=}v'_1 \wedge \cdots \wedge a_n{=}v'_n)[k]$$

so letting

$$A = \{\texttt{caused(c1=v1,k)},\ldots,\texttt{caused(cm=vm,k)},\ \texttt{happens(a1=v1',k)},\ldots,\texttt{happens(an=vn',k)}\}$$

we have that by the definition of M, $A \subseteq M$. Now, by the inductive hypothesis $A \subseteq T^{\omega}_{P^M}(\emptyset)$, so that clearly, the atoms `all_caused([c1=v1,...,cm=vm], k)` and `all_happen([a1=v1',...,an=vn'], k)` are also in $T^{\omega}_{P^M}(\emptyset)$. But that means `caused(c=v,k+1)` $\in T^{\omega}_{P^M}(\emptyset)$ as desired.

Now, suppose there is a law of form (ii), and as stated, no law

$$c[k+1]{=}v' \Leftarrow G$$

in $(\Gamma_t^D)^*$ such that $X \models G$. We first show that `clipped(c=v,k,k+1)` $\notin M$. Suppose for contradiction that `clipped(c=v,k,k+1)` is a member of M. But then it is clear from the definition of M that the presence of this `clipped/3` atom would contravene the stipulation about laws $c[k+1]{=}v' \Leftarrow G$ given above. So `clipped(c=v,k,k+1)` $\notin M$. Now, $X \models c[i]{=}v$ and so `caused(c=v,k)` $\in M$, and thus by the induction hypothesis, `caused(c=v,k)` $\in T^{\omega}_{P^M}(\emptyset)$. We know that `inertial(c=v)` $\in T^{\omega}_{P^M}(\emptyset)$, and so by the fact that `clipped(c=v,k,k+1)` $\notin M$, the definition of the stable model semantics, and the nature of P^M, we have that `caused(c=v,k+1)` is in $T^{\omega}_{P^M}(\emptyset)$ as desired.
Next, suppose we have a law of form (iii), with, as we know, $X \models c[k+1]{=}v$. We also know that there is no $v' \neq v$ such that there is a law

$$c[k+1]{=}v' \Leftarrow G$$

such that $X \models G$. As X is an interpretation, we have for all $v' \neq v$, that $X \not\models c[k+1]{=}v'$, so that by the definition of M, we get for all $v' \neq v$, that `caused(c=v',k+1)` is not in M. However, as $T^{\omega}_{P^M}(\emptyset) \subseteq M$, then for all $v' \neq v$, we have `caused(c=v',k+1)` $\notin T^{\omega}_{P^M}(\emptyset)$. As we clearly have `default(c=v)` $\in T^{\omega}_{P^M}(\emptyset)$, then by the fifth axiom for `caused/2`, we have `caused(c=v,k+1)` $\in T^{\omega}_{P^M}(\emptyset)$, as we wish.
The next stage is a little more complex, and itself requires an inductive proof (within the main one). First define $S^{all} = \{c[k+1]{=}v \mid X \models c[k+1]{=}v\}$, and also

$$\begin{aligned} S' = \{c[k+1]{=}v \mid X \models c[k+1]{=}v \wedge c \in \sigma^f \wedge \\ (\exists \text{ a rule } (i) \text{ in } (\Gamma_t^D)^* \text{ with body } F \text{ and } X \models F \vee \\ \exists \text{ a rule } (ii) \text{ in } (\Gamma_t^D)^* \text{ with body } F \text{ and } X \models F \vee \\ \exists \text{ a rule } (iii) \text{ in } (\Gamma_t^D)^* \text{ with body } F \text{ and } X \models F)\} \end{aligned}$$

We have already shown that $S' \subseteq T^{\omega}_{P^M}(\emptyset)$, and so it remains to show that $S = S^{all} - S'$ is such that $S \subseteq T^{\omega}_{P^M}(\emptyset)$. So consider the set L of rules of the form

$$c[k+1]{=}v \Leftarrow (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[k+1]$$

such that $c[k+1]{=}v \in S$ and $X \models (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[k]$. We define a sequence of sets $E_0, \ldots, E_{j-1}$, as shown in Section 3.1.1 (the presence of the time-stamp $k{+}1$ is clearly not problematic for us). Further, we define a sequence $E_0^*, \ldots, E_{j-1}^*$, by

$$E_i^* = \{c[k+1]{=}v \mid c \in E_i \text{ and } X \models c[k+1]{=}v\}$$

We will use our sequence of sets E_i^* in a proof by induction on i. We claim that the following holds:

$$\forall (c[k+1]{=}v \in \bigcup_{i=0}^{j-1} E_i^*) \ \texttt{caused(c=v,k+1)} \in T^{\omega}_{P^M}(\emptyset) \tag{3.2}$$

Base ($i = 0$): Assume $c[k+1]{=}v \in E_0^*$, so that $c \in E_0$. Then clearly, c depends on no constant in C_L (by Theorem 3.7). Thus evidently either $c[k+1]{=}v$ is at the head of no law in L, or else it is at the head of a law $c[k+1]{=}v \Leftarrow \top$. If the

former, then $c[k+1]{=}v \in S'$, and so `caused(c=v,k+1)` $\in T^{\omega}_{P^M}(\emptyset)$. If the latter, then `causes(c=v,[])` must be a fact in our logic program P^M, and thus clearly as `all_caused([], k+1)` is in $T^{\omega}_{P^M}(\emptyset)$, we have `caused(c=v,k+1)` $\in T^{\omega}_{P^M}(\emptyset)$. So we have our base case.
Induction step: Assume the result for $i \leqslant l$, so that where $c[k+1]{=}v \in E^*_i$ for some $i \leqslant l$, we have `caused(c=v,k+1)` $\in T^{\omega}_{P^M}(\emptyset)$. We will show the result for $i = l+1$.
To that end, let $c[k+1]{=}v \in E^*_{l+1}$. Then there is some law

$$c[k+1]{=}v \Leftarrow (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[k+1]$$

in L with $X \models (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[k+1]$. Clearly, however, c depends on $c_1, \ldots, c_n$, so that by Theorem 3.7, we must have that $c_1, \ldots, c_n$ are in $E_0, \ldots, E_l$, so that by the definition of the E^*_i, the $c_1[k+1]{=}v_1, \ldots, c_n[k+1]{=}v_n$ are in the $E^*_0, \ldots, E^*_l$. Then by our induction hypothesis, we have that `caused(c1=v1,k+1)`, ..., `caused(cn=vn,k+1)` $\in T^{\omega}_{P^M}(\emptyset)$. As we have that `causes(c=v, [c1=v1,...,cn=vn])` $\in T^{\omega}_{P^M}(\emptyset)$, then also `caused(c=v,k+1)` $\in T^{\omega}_{P^M}(\emptyset)$, which is our induction step concluded.
Thus, by induction, we have our result (3.2).
It is clear that

$$S = \bigcup_{i=0}^{j-1} E^*_i$$

so that we have now concluded our (outer) induction step for `caused/2`.
And thus, we have shown by induction that if `caused(c=v,i)` $\in M$, then `caused(c=v,i)` $\in T^{\omega}_{P^M}(\emptyset)$.
`all_caused/2`: The case for an empty list as the first argument is entirely straightforward.
So, let `all_caused([c1=v1,...,cn=vn,i)` $\in M$. Then by the definition of M, $X \models (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[i]$, and thus the atoms `caused(c1=v1,i)`,..., `caused(cn=vn,i)` must all be in $T^{\omega}_{P^M}(\emptyset)$. The desired result immediately follows: `all_caused([c1=v1,...,cn=vn],i)` $\in T^{\omega}_{P^M}(\emptyset)$.
`all_happen/2`: Suppose that `all_happen([a1=v1,...,an=vn], i)` is in M. Then by an argument which directly parallels that of the preceding case, we have `all_happen([a1=v1,...,an=vn], i)` $\in T^{\omega}_{P^M}(\emptyset)$.
`clipped/3`: Assume `clipped(c=v,k,k+1)` $\in M$. We proceed according to the cases which must hold, given the definition of M.
In the first case, we must have a law in D

$$c{=}v' \textbf{ if } \top \textbf{ after } (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m) \wedge (a_1{=}v'_1 \wedge \cdots \wedge a_n{=}v'_n)$$

with $X \models (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m)[k]$, and $X \models (a_1{=}v'_1 \wedge \cdots \wedge a_n{=}v'_n)[k]$. But then, by the definition of M, and according to previous cases of the current result, we have that `caused(c1=v1,k)`,...,`caused(cm=vm,k)` are in $T^{\omega}_{P^M}(\emptyset)$, as are `happens(a1=v1',k)`,...,`happens(an=vn',k)`. So, we must also have that

`all_caused([c1=v1,...,cm=vm], k)`,
`all_happen([a1=v1',...,an=vn'], k)`

are in $T^{\omega}_{P^M}(\emptyset)$. Then, by the definition of `clipped/3` in P^M, we must have that `clipped(c=v,k,k+1)` $\in T^{\omega}_{P^M}(\emptyset)$, as desired.

In the second case, we must have a law

$$c{=}v' \textbf{ if } c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n$$

in D, such that $X \models (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[k+1]$, which means also that we have a fact `causes(c=v, [c1=v1,...,cn=vn])` in $T^{\omega}_{P^M}(\emptyset)$ We must also have `caused(c1=v1,k+1)`,...,`caused(cn=vn,k+1)` in M, and so also in $T^{\omega}_{P^M}(\emptyset)$. Thus clearly, by the definition of `clipped/3`, we get that `clipped(k,k+1,c,v)` is in $T^{\omega}_{P^M}(\emptyset)$, as we want. That concludes the case for `clipped/3`.
`overridden/2`: Assume that `overridden(c=v,i)` is in M. Then by the definition of M, we have that $X \models c[i] = v'$ for some $v' \neq v$. Thus clearly `caused(c=v',i)` $\in M$, and by previous a previous case of the current proof, for this v' with $v' \neq v$, `caused(c=v',i)` $\in T^{\omega}_{P^M}(\emptyset)$. But then evidently `overridden(c=v,i)` $\in T^{\omega}_{P^M}(\emptyset)$ as desired.
That concludes the case analysis for our intermediate result, and so we have shown that $M \subseteq T^{\omega}_{P^M}(\emptyset)$. With the previous half of $(\rightarrow)$, this gives us that $M = T^{\omega}_{P^M}(\emptyset)$, so that M is a stable model of our P.
It remains to check that the four conditions (i)–(iv) are fulfilled. That the first two are satisfied is an immediate consequence of the definition of M. The third is a condition relating to `never/1` predicates. Assume for contradiction that there is some `never([c1=v1,...,cn=vn])` $\in P$, with some i such that `all_caused([c1=v1,...,cn=vn])` $\in M$. Then clearly by the definition of M, we have $X \models (c_1{=}{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[i]$, and as our `never/1` fact was included in P, there must be a law

$$\bot \textbf{ if } c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n$$

in D. But then $\bot \in ((\Gamma^D_t)^*)^X$, so that $X \not\models_{\mathsf{C}} (\Gamma^D_t)^*$, our desired contradiction. The case for the fourth condition on the model M goes through similarly.
So, we have shown $(\rightarrow)$.

$(\leftarrow)$: Now again suppose we have an interpretation X of the signature of $(\Gamma^D_t)^*$, and let M be a stable model of $LP(D, t, Init, \{Haps_i \mid 0 \leqslant i < t\})$ (which we again abbreviate to P), which satisfies conditions (i)–(iv) given in the statement of the current proposition. We must show that $X \models_{\mathsf{C}} (\Gamma^D_t)^*$, ie., that X is the unique model of $((\Gamma^D_t)^*)^X$. That will clearly be the case if the following are fulfilled:

- for all i such that $0 \leqslant i < t$ and all $a \in \sigma^a$, there is exactly one $v \in dom(a)$ such that $a[i]{=}v \in ((\Gamma^D_t)^*)^X$; furthermore, for that v, we have $X \models a[i]{=}v$,
- for all i such that $0 \leqslant i \leqslant t$ and all $c \in \sigma^f$, there is exactly one $v \in dom(c)$ such that $c[i]{=}v \in ((\Gamma^D_t)^*)^X$; furthermore, for that v, $X \models c[i]{=}v$,
- $\bot \notin ((\Gamma^D_t)^*)^X$.

We consider the three requirements in turn.
Thus, let i be such that $0 \leqslant i < t$, and let $a \in \sigma^a$. Then as X is an interpretation of the signature of $(\Gamma^D_t)^*$, there is a unique member of $dom(c)$ to which X assigns a; let that member be v, so that $X \models a[i]{=}v$. Then by condition (ii) on M, we have that v is the only member of $dom(a)$ such that `happens(a=v,i)` $\in M$. Since M is the least Herbrand model of P^M, however, and by the definition of P, we must have that v is the only member of the domain of a such that

$a[i]{=}v \in Haps_i$. But that means, by the definition of $(\Gamma_t^D)^*$, that v is the only member of the domain of a such that there is a causal rule $a[i]{=}v \Leftarrow \top \in (\Gamma_t^D)^*$. Thus it is clear that v is the only member of the domain of a such that $a[i]{=}v \in ((\Gamma_t^D)^*)^X$. Therefore, our first condition is fulfilled, and we progress to the second.

Thus take i for $0 \leqslant i \leqslant t$, and $c \in \sigma^f$. Again, since X is an interpretation, we have a unique $v \in dom(c)$ with $X \models c[i]{=}v$. We must show that $[i]c{=}v \in ((\Gamma_t^D)^*)^X$, and that there is no other $v' \in dom(c)$ such that $c[i]{=}v' \in ((\Gamma_t^D)^*)^X$. We first show the former, then the latter.

Evidently, as $X \models c[i]{=}v$, we have that `caused(c=v,i)` $\in M$ by condition (i) on M. As, by hypothesis, we have that M is the least Herbrand model of P^M, then clearly `caused(c=v,i)` $\in T_{P^M}^{\omega}(\emptyset)$, and also there is a least positive integer such j that `caused(c=v,i)` $\in T_{P^M}^{j}(\emptyset)$. Now, it must be the case that one of the five clauses defining `caused/2` has an instantiation such that the head is `caused(c=v,i)`, and all conjuncts of the body are members of $T_{P^M}^{j-1}(\emptyset)$. We accordingly proceed by a case analysis of the five (types of) clauses in the definition of `caused/2`.

Clause 1: in this case, we must have $i = 0$ and `init(c=v)` $\in T_{P^M}^{j-1}(\emptyset)$. But then clearly `init(c=v)` $\in P$, so that by the definition of P, $c[i]{=}v \in Init$. But then $c[i]{=}v \Leftarrow \top$ is a causal law in $(\Gamma_t^D)^*$, so that clearly $c[i]{=}v \in ((\Gamma_t^D)^*)^X$ as desired.

Clause 2: clearly, we must have some dynamic law

$$c{=}v \textbf{ if } \top \textbf{ after } (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m) \wedge (a_1{=}v_1' \wedge \cdots \wedge a_n{=}v_n')$$

in D such that

$$A = \{\texttt{caused(c1=v1,i-1)}, \ldots, \texttt{caused(cm=vm,i-1)}, \texttt{happens(a1=v1',i-1)}, \ldots, \texttt{happens(an=vn',i-1)}\}$$

are such that $A \subseteq T_{P^M}^{j-1}(\emptyset)$. But then since M is the least Herbrand model of P^M, we must have $A \subseteq M$, and so by conditions (i) and (ii) on M, evidently

$$X \models (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m)[i-1] \wedge (a_1{=}v_1' \wedge \cdots \wedge a_n{=}v_n')[i-1].$$

Thus, given the existence of the dynamic law whose head is $c{=}v$, mentioned above, and the way in which $(\Gamma_t^D)^*$ is formed, clearly we have $c[i]{=}v \in ((\Gamma_t^D)^*)^X$ as desired.

Clause 3: here, we must have some static law

$$c{=}v \textbf{ if } c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n$$

in D such that for

$$A = \{\texttt{caused(c1=v1,i)}, \ldots, \texttt{caused(cn=vn,i)}\}$$

we have $A \subseteq T_{P^M}^{j-1}(\emptyset)$. Then, by condition (i) on M, we get

$$X \models (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[i],$$

and so $c[i]{=}v \in ((\Gamma_t^D)^*)^X$, as desired.

Clause 4: as `inertial(c=v)` $\in T_{P^M}^{*}(\emptyset)$, we have that **inertial** $c{=}v$ must be a law of D, and thus clearly there is a causal law

$$c[i]{=}v \Leftarrow c[i]{=}v \wedge c[i-1]{=}v$$

in $(\Gamma_t^D)^*$. However, as the body of the (instantiation of the) fourth clause is required to be true, we also have that `caused(c=v,i-1)` $\in T_{P^M}^{j-1}(\emptyset)$, and so `caused(c=v,i-1)` $\in M$, which by condition (i) on M means that $X \models c[i-1]{=}v$. But then we have $X \models c[i]{=}v \wedge c[i-1]{=}v$, so that $c[i]{=}v \in ((\Gamma_t^D)^*)^X$ as required.
Clause 5: we know for this that `default(c=v)` $\in D$, which gives us that there is a causal law

$$c[i]{=}v \Leftarrow c[i]{=}v$$

in $(\Gamma_t^D)^*$. It is immediate that $c[i]{=}v \in ((\Gamma_t^D)^*)^X$.
So, we have shown the first half of what we need, that where $X \models c[i]{=}v$, then $c[i]{=}v \in ((\Gamma_t^D)^*)^X$. It remains to show that there is no other $v' \in dom(c)$ for which $c[i]{=}v' \in ((\Gamma_t^D)^*)^X$.
Thus, suppose for contradiction that there is $c[i]{=}v' \in ((\Gamma_t^D)^*)^X$, for $v' \neq v$. Then there must be some causal law $c[i]{=}v' \Leftarrow F$ in $(\Gamma_t^D)^*$ such that $X \models F$. A little thought shows that such a causal law cannot stem from an **inertial** or **default** law of D, nor from any of the sets $Init$ or $Haps_i$. Thus the law must be a 'regular' static or dynamic law of D (i.e., one that is not an expression of default or inertial behaviour), and is thus of one of the forms:

- $c[i]{=}v' \Leftarrow c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n$, or
- $c[i]{=}v' \Leftarrow (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m)[i-1] \wedge (a_1{=}v_1' \wedge \cdots \wedge a_n{=}v_n')[i-1]$.

In the first case, we have $X \models (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m)[i]$, and so by the first condition on M, and letting A be $\{$`caused(c1=v1,i)`$,\ldots,$`caused(cm=vm,i)`$\}$, we have $A \subseteq M$. But as $M = T_{P^M}^{\omega}(\emptyset)$, then there must be some least positive integer j such that $A \subseteq T_{P^M}^{j}(\emptyset)$. Now,it is evident that `caused(c=v',i)` $\in T_{P^M}^{j+2}(\emptyset)$, and `caused(c=v',i)` $\in M$. However, by the definition of M, we have that $X \models c[i]{=}v'$, which is impossible as $v' \neq v$ and $X \models c[i]{=}v$. So we have a contradiction.
Thus, suppose instead that there is a causal law of the second of the forms enumerated above, and that we have $X \models (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m)[i-1]$ and $X \models (a_1{=}v_1' \wedge \cdots \wedge a_n{=}v_n')[i-1]$. Let

$$A = \{\texttt{caused(c1=v1,i-1)},\ldots,\texttt{caused(cm=vm,i-1)},\\ \texttt{happens(a1=v1',i-1)},\ldots,\texttt{happens(an=vn',i-1)}\}$$

so that by the first and second conditions imposed on M, we have $A \subseteq M$. So, $A \subseteq T_{P^M}^{\omega}(\emptyset)$, and there is again a least integer j such that $A \subseteq T_{P^M}^{j}(\emptyset)$. It is obvious that `all_happen([a1=v1',...,an=vn'], i-1)` $\in T_{P^M}^{j+n}(\emptyset)$ and for similar reasons, that `all_caused([c1=v1,...,cm=vm], i-1)` $\in T_{P^M}^{j+m}(\emptyset)$, so letting p be the greater of m and n, both these atoms are members of $T_{P^M}^{j+p}(\emptyset)$. It is therefore clear that, given the nature of the third clause defining `caused/2`, we must have `caused(c=v',i)` $\in T_{P^M}^{j+p+1}(\emptyset)$, so that `caused(c,v',i)` $\in M$, and thus as before $X \models c[i]{=}v'$, which is a contradiction.
So, under the supposition of $X \models c[i]{=}v$, we have shown that there can be no other $v' \in dom(c)$ such that $c[i]{=}v' \in ((\Gamma_t^D)^*)^X$.
That completes the second of our main requirements for showing $(\leftarrow)$. We must now prove that $\bot \notin ((\Gamma_t^D)^*)^X$.
So, suppose for contradiction that $\bot \in ((\Gamma_t^D)^*)^X$. This can only happen in two ways:

- there is $\bot$ **if** $c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n$ in D, with for some i such that $0 \leqslant i \leqslant t$, $X \models (c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n)[i]$, or
- there is $\bot$ **if** $\top$ **after** $(c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m) \wedge (a_1{=}v'_1 \wedge \cdots \wedge a_n{=}v'_n)$ in D, with for i such that $0 \leqslant i < t$, $X \models (c_1{=}v_1 \wedge \cdots \wedge c_m{=}v_m)[i]$ and $X \models (a_1{=}v'_1 \wedge \cdots \wedge a_n{=}v'_n)$.

In the first case, let A stand for $\{$`caused(c1=v1,i)`$, \ldots,$ `caused(cn=vn,i)`$\}$. By the definition of M, we get $A \subseteq M$. Yet as `never([c1=v1,...,cn=vn])` $\in M$, condition (iii) on the definition of M is broken, yielding a contradiction.
Alternately, suppose there is a dynamic law with $\bot$ as its head, according to the above. Then letting

$$A = \{\texttt{caused(c1=v1,i)}, \ldots, \texttt{caused(cm=vm,i)}, \\ \texttt{happens(a1=v1',i)}, \ldots, \texttt{happens(an=vn',i)}\}$$

it is obvious that $A \subseteq M$ by the definition of M, so that there is some least j for which $A \subseteq T^j_{P^M}(\emptyset)$. Now, let p be the greatest of m and n, and so also `all_caused([c1=v1,...,cm=vm],i)` $\in T^{j+p}_{P^M}(\emptyset)$ and we also have `all_happen([a1=v1',...,an=vn'],i)` $\in T^{j+p}_{P^M}(\emptyset)$. It now obviously follows that these two `all_caused/1` and `all_happen/2` atoms are in M, which together with the presence of

```
nonexecutable([a1=v1',...an=vn'], [c1=v1,...,cm=vm]).
```

contradicts condition (iv) on the definition of M.
Thus, we have derived a contradiction from the supposition that $\bot \in ((\Gamma^D_t)^*)^X$, and so the third of our main conditions on the direction $(\leftarrow)$ of our proof is shown.
And with it our result. ⌟

3.4 Consistency and Models

Theorem 3.10 guarantees that, where X is an interpretation of the signature of an action description D of $\mathcal{EC}+$, then X is a model of Γ^D_t iff X corresponds to a stable model of the logic programs we describe; models of the causal theories and logic programs completely agree in the answers they give to appropriate queries. Yet an important question remains. Since our objective here is to work entirely within a logic-programming framework, how can we be sure that the programs $LP(D, t, Init, \{Haps_i \mid 0 \leqslant i < t\})$ which we use determine as their stable models sets of atoms which correspond to interpretations of σ_t? And if a given logic program, complete with information about the initial state and narrative of events, does have as its stable models only sets which correspond to interpretations, how can we be sure that we do not fall foul of the information contained in facts of the predicates `never/1` and `nonexecutable/2`?

These questions are pressing. As the objective in using logic programs is to be able to answer queries about the systems defined in a piecemeal fashion, without constructing entire runs through the transition system in the way CCalc does, we cannot simply look at the content of the entire stable model, and check whether it represents an interpretation of σ_t and breaks no constraint imposed by `never/1` and `nonexecutable/2`. Our logic programs can use SLDNF to

answer queries, and though we have proved correctness of the implementation based on a declarative semantics of stable models, in practice we will have little to do with those stable models.

Given the relation between $\mathcal{EC}+$ and the event calculus (which we will explore more thoroughly in Section 3.10), it is not surprising that the same issue occurs with that formalism. Reasoning tasks in event calculi of the logic-programmed sort are typically of two kinds: that species of answering questions about the values of individual fluents, typified in a PROLOG query

```
?- holds_at(c=v, t).
```

and also the reasoning which ought to be performed before this particular querying—that which determines whether a narrative of events is possible and consistent. The latter task can be seen as a matter of checking whether a number of integrity constraints are satisfied. For instance, if information about a domain (the initial state, laws governing the initiation of fluents, the narrative of events) is to imply a consistent narrative, then there should be no point in the narrative at which the performance of an action causes the same fluent to have different values. Consider a domain with Boolean action constant a and fluent constants p and q, and where the effects axioms, initial state and narrative of events are set by the following clauses:

```
initially(p=tt).
initially(q=tt).

happens(a=tt, 0).

initiates(a=tt, q=tt, T) :-
  holds_at(p=tt, T).

initiates(a=tt, q=ff, T) :-
  holds_at(p=tt, T).
```

In conjunction with the axioms for the event calculus we presented in Section 2.4 (and assuming appropriate groundings), there are certainly stable models for the above. Yet no stable model contains an atom `holds_at(q=V,1)` for any value of `V`, and it is clear that this should be attributed to an inconsistency in the event calculus program.

A precisely analogous problem occurs with $\mathcal{EC}+$, as it is not hard to imagine: for even given complete and consistent (in the sense defined in Section 3.2.3) information about an initial state and narrative, an action description with the laws

$$\textbf{caused } p \textbf{ if } \top$$
$$\textbf{caused } \neg p \textbf{ if } \top$$

will determine a logic program which has stable models, but which does not correspond to any interpretation of σ_t.

So, it remains accurately to define the circumstances in which the logic programs we defined in Section 3.2 fail to determine interpretations of σ_t, and then to describe ways in which we can ensure that our initial states and narratives

can be shown to imply a consistent path through the transition system defined in $\mathcal{EC}+$.

Let D be an action description of $\mathcal{EC}+$ and σ its signature, and suppose we are given some $LP(D, t, Init, \{Haps_i \mid 0 \leqslant i < t\})$; call this program P. If M is a stable model of P, we must check that

- for all $i \leqslant t$ and $c \in \sigma^f$, there is an atom `caused(c=v,i)` $\in M$;
- for all $i \leqslant t$ and $c \in \sigma^f$, there is at most one atom `caused(c=v,i)` $\in M$;
- for all atoms `never(F)` $\in M$, we do not have `all_caused(F,i)` $\in M$ for any $i \leqslant t$;
- for all atoms `nonexecutable(A,F)` $\in M$, we do not have both that `all_happen(A,i)` $\in M$ and `all_caused(F,i)` $\in M$, for $i < t$.

We consider these checks in turn.

First, it must be true that every fluent constant is caused to have a value in every state of the run. Constraints which we have imposed on the specification of the initial state (which is determined by the set *Init*) mean that for $i = 0$, there is at least one value v of c such that `caused(c=v,0)` $\in M$. Now, for a given constant c, if there is a law

inertial c

in D, then at every later time i there *must* be an atom `caused(c=v,i)` in M for some value $v \in dom(c)$, as a simple argument based on the structure of the axioms of P shows. (Indeed, it should be intuitively obvious that if c is caused to have a value initially, and if the value of c is caused to persist by default, then c will always be caused to have a value.) So our check that all fluent constants c are caused to have a value at all times of a run, reduces to the check that all *non-inertial* constants are caused to have such a value. Similar reasoning reduces the problem to that of determining that all non-inertial constants for which there is no law

default c=v

in D are caused to have a value.

So, we must check that for all c such that neither

inertial c nor **default** c=v

(for some $v \in dom(c)$), and all i with $0 < i \leqslant t$, there is some `caused(c=v,i)` $\in M$. Checks of various degrees of sophistication can be made here. The most crude, but simple to implement, is to proceed from $i = 1$ to $i = t$ successively, checking each constant c in turn. We will not describe how to do this in detail, except to say that it is convenient in this process to make use of the sets E_j which were defined in Section 3.1.1: we can proceed from the constants which depend on no constant occurring in a static law of D, upwards.

Alternative checks, of greater sophistication, are possible, based on an analysis of the causal laws of the action description. If, for instance, there is a fluent constant c whose behaviour is not governed by inertial laws or default conditions, we can look to see whether a value for c is determined by static or dynamic laws, one of which must hold in every possible state or across every

possible transition of the system. As an example, consider an action description where a Boolean c is free from inertia and default determination as described, and where there are static laws

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

in the action description. Suppose too that q is Boolean. It is clear that at least one of the bodies of the above laws must be true in every state of the transition system, and so in every state, p is determined to have a value. The general principle here is that where c is a constant we wish to ensure has a value, and where there are causal laws

$$\begin{aligned}
&\textbf{caused } c{=}v_1 \textbf{ if } F_1,\\
&\quad\vdots\\
&\textbf{caused } c{=}v_m \textbf{ if } F_m,\\
&\textbf{caused } c{=}v_{m+1} \textbf{ if } F_{m+1} \textbf{ after } G_1,\\
&\quad\vdots\\
&\textbf{caused } c{=}v_{m+n} \textbf{ if } F_{m+n} \textbf{ after } G_n,
\end{aligned}$$

such that, for every transition (s, e, s') of the transition system, either $s' \models F_i$ for some i such that $1 \leqslant i \leqslant m$, or else $s \cup e \models G_k$ and $s' \models F_{m+k}$ for some k with $1 \leqslant k \leqslant n$, then we have shown that c is caused to have a value in every state of the run.

Checking that these conditions hold can be more or less complicated; perhaps the most simple case is represented above, where there are static laws, one of whose bodies, it can immediately seen, must hold in every state. More searching analyses could take into account the information given about the action components e of transitions. This form of analysis may often be quicker than moving through a given run, checking that each constant c is caused to have a value across every transition. (In practice, we have limited ourselves to the most straightforward kinds of check, as epitomized in the example with static rules, and combined this with brute-force checking for constants not covered.)

The second check we must perform is to verify that fluent constants of the action description are not caused to have more than one value in states later than the first. This situation arises when given a preceding state s and actions e, no state s' can satisfy

$$(\Gamma_1^D)^{s[0]\cup e[0]\cup s'[1]}$$

and where the reason for this is *not* because $\bot$ is contained in this reduct, but because all such reducts would require the same constant $c[1]$ to have two different values. With definite action descriptions (with which we are exclusively concerned in $\mathcal{EC}+$), this reduces to there being a pair of causal laws whose bodies are both true across the transition: two fluent dynamic causal laws

$$c{=}v_1 \textbf{ if } \top \textbf{ after } A_1 \wedge F_1 \qquad \text{and} \qquad c{=}v_2 \textbf{ if } \top \textbf{ after } A_2 \wedge F_2$$

such that $s \models F_1 \wedge F_2$ and $e \models A_1 \wedge A_2$; or two static laws

$$c{=}v_1 \textbf{ if } F_1 \qquad \text{and} \qquad c{=}v_2 \textbf{ if } F_2$$

such that the bodies F_1 and F_2 would be required to be true in any succeeding s'; or laws

$$c{=}v_1 \textbf{ if } \top \textbf{ after } A_1 \wedge F_1 \qquad \text{and} \qquad c{=}v_2 \textbf{ if } F_2$$

where $s \models F_1$ and $e \models A_1$, and where any succeeding s' would be constrained to have $s' \models F_2$ (in all cases, $v_1 \neq v_2$). As before, simple cases of this verification are easy to enumerate: if, for instance, our input narrative has, at a given time i, $e[i] \models a[i]{=}v'$, and there are laws

$$p \textbf{ if } \top \textbf{ after } a{=}v' \qquad \text{and} \qquad \neg p \textbf{ if } \top \textbf{ after } a{=}v',$$

then clearly the input narrative cannot represent a run through the transition system defined by D, for this run breaks down at time i. As previously, this checking may not always reduce to such simple cases, and we are either forced to be sophisticated in our analyses or to employ a constant-by-constant brute-force method.

Finally, we need to ensure that $\bot$ is not caused. It could be so in two ways: through a fluent dynamic or a static causal law. In the former case, there will be a law

$$\textbf{nonexecutable } A \textbf{ if } F$$

in the action description, where F is made true by the fluent constants at some step i of a run, and A is made true by the action constants at the same time. Where F is empty, and the law has the form $\bot$ **if** A, then the check is easy to make; otherwise we need an inductive process which starts at time 0 and moves incrementally through the run, checking that F is not satisfied at any time. In the latter case, with a static rule whose head is $\bot$, the same process can be used.

As an example, consider the 'farmyard' domain (see Section 2.1.7), with the initial state and narrative of actions here:

```
init(alive(bill)=tt).          init(loaded=ff).
init(alive(turkey)=tt).        init(smiling(bill)=ff).
init(loc(bill)=house).         init(smiling(turkey)=tt).
init(loc(turkey)=barn).        init(target=none).

happens(load=tt, 0).
happens(aim=field, 1).
happens(walk(bill)=field, 2).
happens(shoot=tt, 3).
happens(walk(turkey)=house, 4).
happens(load=tt, 6).
happens(walk(turkey)=barn, 7).
happens(miracle(bill)=tt, 8).
happens(shoot=tt, 9).
happens(walk(bill)=house, 10).
happens(miracle(turkey)=tt, 11).
```

(This is the same initial information and narrative of actions we use in Section 3.6). Now, the set *Init* clearly represents an initial state, as each fluent constant is assigned a single value, and the interpretation is consistent with the

static laws of the action description: those static laws are

$$alive(x) \textbf{ if } smiling(x)$$
$$\neg smiling(x) \textbf{ if } \neg alive(x)$$

(for $x \in \{Bill, Turkey\}$), and the only body of one of these made true in the initial state is $smiling(Turkey)$; since we also have $alive(Turkey) \in Init$, the atoms must form a state.

The next stage is to check the narrative, and this—given the exogeneity of all action constants—is simply a matter of verifying that each such constant has a unique value at every time of the narrative. Given the way in which our implementation of $\mathcal{EC}+$ automatically fills out partial narratives, in line with default values for action constants, to full interpretations of σ_t^a, we immediately have that the events above determine a unique value for each member of σ_t^a.

Since there is a law **inertial** c for each fluent constant c in the signature, we are assured that the fluent constants are all caused to have values. In order to see whether those caused values are, for each fluent constant at each time, unique, we first find all pairs of causal laws which might be true across transitions, where the heads of the two laws assign different values to the same constant. There are 21 such pairs in the action description, three of which are

caused $loc(Turkey)$=*field* **if** $\top$ **after** $walk(Turkey)$=*field*
caused $loc(Turkey)$=*house* **if** $\top$ **after** $walk(Turkey)$=*house*

caused *target*=*house* **if** $\top$ **after** *aim*=*house*
caused *target*=*none* **if** $\top$ **after** *load*

caused $\neg smiling(Turkey)$ **if** $\neg alive(turkey)$
caused $smiling(turkey)$ **if** $\top$ **after** $miracle(turkey) \wedge \neg alive(turkey)$

Elementary checking can show that the first pair of causal laws, both fluent dynamic, presents no problem: regardless of the initial state and narrative of actions, the bodies of any of the pairs of causal rules which result from these laws (the pairs for each time-stamping, in Γ_{12}^D) can never both be true. This is obvious, given that the laws' bodies are conjunctions which exclude each other by the presence of the same action constant $walk(turkey)$ in each, associated with a different value. The second pair of laws cannot be seen to be safe immediately, apart from the details of the initial state and event narrative. Yet in fact, if we consider the full narrative of events implied in our input, we see that *aim*=*house* and *load* are never true together, across any of the twelve transitions forming our run. So the second pair of laws is safe.

Verifying that the third pair of laws does not yield causal rules whose bodies are jointly satisfied is more involved. Inspection of the bodies of the laws shows that the bodies themselves do not exclude each other in the manner of the first pair—indeed, as we are dealing with a static law and fluent dynamic law, this would be impossible. Similarly, the narrative of events cannot remove all threats, because

```
happens(miracle(turkey)=tt, 11)
```

is part of our narrative. Although other transitions of the run are safe—because the full narrative implied by our input events makes *miracle*(*Turkey*) true at no other time—more work is necessary to show that conflicting values for *smiling*(*Turkey*) are not caused in the state at time 12. This is either a matter of proving that $\neg$*alive*(*Turkey*) one of the bodies of the pair must be false, which amounts to proving that *alive*(*Turkey*) must be true either at time 11 or 12. We do this by methods described above.

Finally, we must show that $\perp$ is not caused. This would occur when there are fluents F and (possibly) actions A which are implied (at some time i) by the partial narrative we submit as input, but where there is a law

nonexecutable A **if** F or **caused** $\perp$ **if** F

in the action description. In the run of the farmyard example we are considering, there are no static laws having $\perp$ as their head; there are, once groundings have been made, 12 **nonexecutable** laws in the action description, and as there are 12 transitions to consider (the run is of length 12), that gives 144 cases to verify. Of these, 134 can be pruned easily because the action component of the transition does not make the relevant *walk*(x)=l action true. The others must be checked by moving through the supposed run from the initial state to the final state, verifying that across any transition at which walk(x)=l is true, both *loc*(x)=l and $\neg$*alive*(x) are false. That can be done quickly, using the axioms of $\mathcal{EC}+$.

It will frequently be the case that we have an intuitive confidence that the action descriptions D we write down, and the information about initial states and narratives of events we supply, do determine stable models of our logic programs which represent interpretations of the signature Γ_t, for a t representing the length of run through the system. In simple cases the checking for consistency we have described above will be unnecessary. But action descriptions of $\mathcal{EC}+$ are often complicated, and may determine interactions which are not wholly foreseeable; in these circumstances the checking becomes of great importance.

3.5 Implementation

We have written a program which enables one to input action descriptions of $\mathcal{EC}+$, together with specifications of an initial state $Init$, and of a narrative of events $Haps_0, \ldots, Haps_{t-1}$ leading up to some maximal time t. Then, one can pose queries to the system about which fluents are caused to have which values in different states of the (implicit) transition system. The program is written in Prolog, and will run on Linux under SICStus[2] (tested under version 3.9.1 and later), SWI-Prolog[3] (tested under 5.2.8 and later) and YAP-Prolog[4] (4.4.3 and later). A number of other features are included in the program, including the ability to write the transition systems defined by the action descriptions to file, in a format which can be processed by the `dot` graph-drawing program (which is part of a suite of open source programs known as Graphviz[5]). We have found

[2] `http://www.sics.se/sicstus/`
[3] `http://www.swi-prolog.org/`
[4] `http://www.ncc.up.pt/~vsc/Yap/`
[5] `http://www.research.att.com/sw/tools/graphviz/`

the latter to be a very useful visualization tool when working with domains of $\mathcal{EC}+$.

Another very useful feature is the way in which actions forming part of a narrative can be represented implicitly, without needing to be written down in full (this was alluded to earlier, in Section 3.2.3). For consider a Boolean action description of $\mathcal{EC}+$ in which there are many action constants, and the runs we are modelling are long. The number of action constants evaluated to the Boolean value **t** at any given time may be low, and it would be long-windedly inconvenient to specify all the instances, over a long run, where an action constant a is evaluated to **f**. For this reason we include in our programs government of the default values of these constants. Let t be the length of run in which we are interested (so that there must be interpretations of $\sigma^a[i]$ for $0 \leqslant i < t$). If, for given $a \in \sigma^a$ and i, there is no atom

```
happens(a=v,i).
```

for any $v \in dom(a)$ in our logic programs, and $\mathbf{f} \in dom(a)$, then we assume that $a[i]$=**f** is intended. If, on the other hand, $\mathbf{f} \notin dom(a)$ but $none \in dom(a)$, we assume that $a[i]$=*none* is intended. Where an action constant a has possible values **f** and *none*, these are usually intended to represent that the action a is not 'performed' (where a denotes the action of some agent), or else does not 'occur' (where a is an event for which no agent is responsible). The sample narrative which we present for the 'farmyard' domain in Section 3.6 makes use of these defaults, and enables the narrative to be presented very concisely, given the largely Boolean domain and sparse occurrence of actions which evaluate to anything other than **f** or **none**.

The way in which what can be called 'partial narratives' are filled out into complete information about actions performed, to satisfy our requirements on completeness, is customizable. If desired, the user may set a value other than **f** to be the default, or may set different values for different action constants.

Some variants of the Event Calculus use PROLOG negation by failure to accomplish the same end of filling out partial narratives into the implicitly defined full narrative of events. In particular, many variants insist on a Boolean signature, albeit implicitly defined, where the occurrence of an action a at time i is represented by the presence of a fact

```
happens(a, i).
```

in the logic program. Whenever such an atom does not occur, it is assumed that the action a does not occur either.

3.5.1 Queries and Explanatory Traces

Given as input a representation of an action description D of $\mathcal{EC}+$, together with a description of the initial state and a narrative, which mentions the performance of actions up to some time $t-1$, several types of query are supported by our implementation:

- `q(c=v, i)`: calling this, for some $c \in \sigma$, elicits an answer from PROLOG whether or not there is a cause for the fluent c to have value v at time i, with $0 \leqslant i \leqslant t$. The argument c may contain a variable (though not itself be one), the argument v may be or contain a variable, and the argument

i may be a variable. In these cases, PROLOG will output all bindings of the variables for which fluent c is caused to have value v at time i;

- `q([c1=v1,i1,...,cn=vn,in])`: check whether, for $1 \leqslant j \leqslant n$, c_j is caused to have value v_j at i_j. Variables may be included as for the first type of query;
- `narrate`: outputs the entire narrative and information on states;
- `narrate(file)`: the same as the previous, but outputs to the file `file`;
- `trans(flag)`: writes the transition system defined by the input action description to file, and then runs `dot` on the file to make a PostScript version. If `flag` is `y`, then `gv` is started on the file written.

Further, each of the queries described above, of the predicates `q/1` and `q/2`, may take an additional, final argument of the form `d(k)`, with `k` a non-negative integer. These augmented queries, in addition to doing the work of the original `q/1` and `q/2`, exploit the style of computation we have implemented in order to give an explanation of why any computed answer *is* an answer to the query. The integer `k` represents the amount of detail required in the explanation; more precisely, `k` is the depth of recursion through the axioms of our logic program after which we stop printing explanatory information. In this way, a query `q(c=v,i,d(0))` would ask for no explanatory information and thus be equivalent to a query `q(c=v,i)`; whereas if `c=v` holds at time `i` because of some causal law

$$\textbf{caused } c{=}v \textbf{ if } \top \textbf{ after } a{=}\mathbf{t} \wedge p{=}\mathbf{f}$$

with `happens(a=tt,i')` (where `i` is the successor of `i'`) as part of the input narrative and `p=ff` established to be true at `i'`, then a query of `q(p=V, 1, d(1))` would give:

```
?- q(p=V, 1, d(1)).

1: p=1 - dynamic law -
p=1 if true after a=tt & p=ff

V = 1
```

The information here represents that the implementation has shown $p[1]{=}1$ by using the fluent dynamic law above. More detailed examples are shown in Section 3.6.

It may be obvious that, in an important sense, explanations for some fluent's being caused may not be unique. Suppose, for example, that all fluents are inertial, and there is some static law

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

in an action description. In some model X, if we have $X \models c'[i]{=}v'$ and $X \models c'[i+1]{=}v'$ (and thus also $X \models c[i]{=}v$ and $X \models c[i+1]{=}v$), then what is the 'explanation' for $X \models c[i+1]{=}v$? Is it caused by an inertial carrying-through of the value of c at i, or as a ramification of c' at $i+1$? In fact, both are plausibly viewed as explanations. In our implementation, if there is such an

overdetermination of causes for a fluent's value, we only pick one explanation to display, and *which* of the various possible explanations we chose has been determined by the order of axioms in the logic programs for $\mathcal{EC}+$. Thus, amongst the four possible explanations for a fluent atom's holding, explanations resting on dynamic laws, static laws, inertia, or default values, will be chosen in that order. In practice this seems to us to give very helpful results.

This feature of multiple causes for a given fluent is discussed again in Section 3.7, with reference to measures for avoiding the recomputation of values for fluents which we have implemented.

3.6 Example—the Farmyard

As an example, we show some queries of the 'Farmyard Resurrection' domain, which we presented as a $\mathcal{C}+$ action description in Section 2.1.7. It can be seen that this action description also qualifies as an $\mathcal{EC}+$ domain. The input file for the entire domain is shown in Appendix A. Let D_{frm} be the name of the $\mathcal{EC}+$ action description.

The narrative and initial conditions are represented by the following facts:

```
init(alive(bill)=tt).          init(loaded=ff).
init(alive(turkey)=tt).        init(smiling(bill)=ff).
init(loc(bill)=house).         init(smiling(turkey)=tt).
init(loc(turkey)=barn).        init(target=none).

happens(load=tt, 0).
happens(aim=field, 1).
happens(walk(bill)=field, 2).
happens(shoot=tt, 3).
happens(walk(turkey)=house, 4).
happens(load=tt, 6).
happens(walk(turkey)=barn, 7).
happens(miracle(bill)=tt, 8).
happens(shoot=tt, 9).
happens(walk(bill)=house, 10).
happens(miracle(turkey)=tt, 11).
```

We give a record of a sample session with $\mathcal{EC}+$, where the following questions are asked:

- is *Bill* alive at time 7?
- what are the locations of all agents at time 10?
- at which times are *Bill* and the *Turkey* at the same location?

We have omitted, for clarity, the usual PROLOG response `No` to a query for which all answers have already been computed.

```
?- q(alive(bill)=tt, 7).
```

```
?- q(loc(X), Y, 10).

X = turkey
Y = barn ;

X = bill
Y = field ;

?- q([loc(bill)=X, T, loc(turkey)=X, T]).

X = field
T = 7 ;

?-
```

The results should be clear: where

$$X = s_0[0] \cup e_0[0] \cup s_1[1] \cup e_1[1] \cup \cdots \cup e_{11}[11] \cup s_{12}[12]$$

is a model of the causal theory $\Gamma_{12}^{D_{frm}}$ which satisfies the input narrative and initial state, we have:

$$X \not\models \mathit{alive}(\mathit{bill})[7]$$
$$X \models \mathit{loc}(\mathit{turkey})[10]{=}\mathit{barn} \wedge \mathit{loc}(\mathit{bill})[10]{=}\mathit{field}$$
$$\{(x,t) \mid X \models \mathit{loc}(\mathit{bill})[t]{=}x \wedge \mathit{loc}(\mathit{turkey})[t]{=}x\} = \{(\mathit{field}, 7)\}$$

Or, in words: *bill* is dead at time 7; at time 10 *turkey* is in the *barn* and *bill* in the *field*; and the only the time at which *bill* and *turkey* are at the same location is time 7, when they meet in the *field*.

The next query asks for an explanation, effectively tracing the course of computation. Thus, it is asked whether *Bill* is caused to be dead at time 5—and if he is so caused, an explanation is to be given, to a level of 6. Fluents are given as output, together with the times at which they are caused, followed by a short description of the means of their causation (the axiom which was used to prove them). If a dynamic or a static law was used in their proof, then it is printed. Then, on the next line down and at an indentation, the manner of causation of the conditions for the dynamic or static law are shown—with the condition for inertial persistence being the fluent in question holding at the time immediately preceding. The level of indentation then corresponds to the number k—in the current instance, $k = 6$. The output itself should aid understanding:

```
?- q(alive(bill)=ff, 5, d(6)).

5: alive(bill)=ff - inertially
    4: alive(bill)=ff - dynamic law -
       [ alive(bill)=ff if true after shoot=tt &
                                      loaded=tt &
                                      target=field &
                                      loc(bill)=field ]
        3: happens(shoot=tt)
        3: loaded=tt - inertially
```

```
            2: loaded=tt - inertially
                1: loaded=tt - dynamic law -
                   [ loaded=tt if true after load=tt ]
                    0: happens(load=tt)
        3: target=field - inertially
            2: target=field - dynamic law -
               [ target=field if true after aim=field ]
                1: happens(aim=field)
        3: loc(bill)=field - dynamic law -
           [ loc(bill)=field if true after walk(bill)=field ]
            2: happens(walk(bill)=field)

Yes
?-
```

The queries entered above referred to particular fluents and variables, possibly with variables. To view the entire narrative, as already described, `narrate` is used; the output has been formatted to resemble that of CCalc, and constants taking the values **f** or *none* have not been printed.

```
?- narrate.

0: smiling(turkey)
0: loc(bill)=house
0: loc(turkey)=barn
0: alive(bill)
0: alive(turkey)

ACTION:  load

1: smiling(turkey)
1: loc(bill)=house
1: loc(turkey)=barn
1: alive(bill)
1: alive(turkey)
1: loaded

ACTION:  aim=field

2: smiling(turkey)
2: loc(bill)=house
2: loc(turkey)=barn
2: alive(bill)
2: alive(turkey)
2: target=field
2: loaded

ACTION:  walk(bill)=field

3: smiling(turkey)
```

```
3: loc(bill)=field
3: loc(turkey)=barn
3: alive(bill)
3: alive(turkey)
3: target=field
3: loaded

ACTION:  shoot

4: smiling(turkey)
4: loc(bill)=field
4: loc(turkey)=barn
4: alive(turkey)
4: target=field

ACTION:  walk(turkey)=house

5: smiling(turkey)
5: loc(bill)=field
5: loc(turkey)=house
5: alive(turkey)
5: target=field

ACTION:

6: smiling(turkey)
6: loc(bill)=field
6: loc(turkey)=house
6: alive(turkey)
6: target=field

ACTION:  load  walk(turkey)=field

7: smiling(turkey)
7: loc(bill)=field
7: loc(turkey)=field
7: alive(turkey)
7: loaded

ACTION:  walk(turkey)=barn  aim=barn

8: smiling(turkey)
8: loc(bill)=field
8: loc(turkey)=barn
8: alive(turkey)
8: target=barn
8: loaded

ACTION:  miracle(bill)
```

```
9: smiling(bill)
9: smiling(turkey)
9: loc(bill)=field
9: loc(turkey)=barn
9: alive(bill)
9: alive(turkey)
9: target=barn
9: loaded

ACTION:  shoot

10: smiling(bill)
10: loc(bill)=field
10: loc(turkey)=barn
10: alive(bill)
10: target=barn

ACTION:  walk(bill)=house

11: smiling(bill)
11: loc(bill)=house
11: loc(turkey)=barn
11: alive(bill)
11: target=barn

ACTION:  miracle(turkey)

12: smiling(turkey)
12: smiling(bill)
12: loc(bill)=house
12: loc(turkey)=barn
12: alive(bill)
12: alive(turkey)
12: target=barn

Yes
?-
```

3.7 Other Measures to Increase Efficiency

A further consequence, beyond that of the need to consider only fluents and actions causally relevant to the truth of fluent constants when answering queries, of finding an implementation for $\mathcal{EC}+$ which uses a logic-programming language such as PROLOG, is that we can make use of the logic-programmer's standard bag of tricks for improving efficiency.

The actual PROLOG implementation of $\mathcal{EC}+$ we have developed employs several methods for reducing redundancy in the computation of answers to queries, which we will describe in this section. The first is the device of *tabling*: storing computed answers to queries, so that when the same query is posed later in the

PROLOG session, answers may be retrieved directly from the database rather than recomputed. Our implementation of tabling has been derived from that of Azevedo in his Ph.D. thesis [dSA95]. The second method we use, related to tabling but solving a slightly different problem of redundancy, is that of checking whether variables present in the top-level query of `caused/3` are repeatedly grounded in identical ways.

The usefulness of both methods can easily be seen, even before we describe their implementation in detail. In $\mathcal{EC}+$ action descriptions there is a recurrent phenomenon of what may be termed *causative overkill*, which occurs when a constant of the signature is caused to have its value in several different ways. Consider, as an example,the Boolean action description with signature $\sigma^f = \{p, q\}$, $\sigma^a = \{a\}$, shown in Figure 3.2. One transition of this action description,

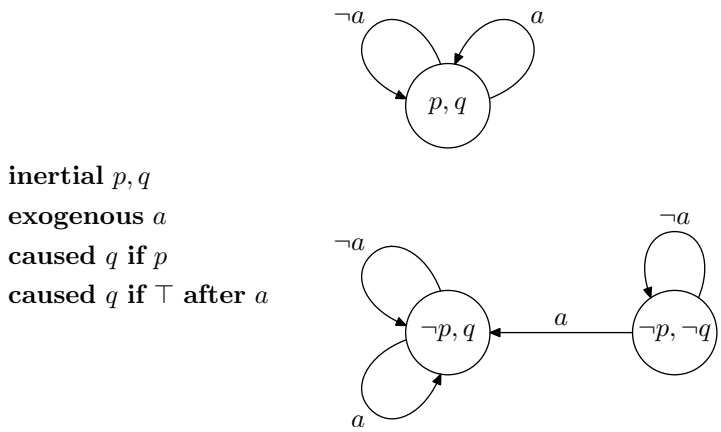

Figure 3.2: System for causative overkill

as can be seen in the diagram, is

$$(\{p, q\}, \{a\}, \{p, q\})$$

Yet the question of what the reason is for q's holding in the successor state of that transition has three plausible answers: through inertia, because of the static law and as p holds in the same state, and also because of the fluent dynamic law. These three ways of answering the question of why q is caused to be true in the transition above, correlated with the presence of causal laws in the action description, are also reflected in the axioms of $\mathcal{EC}+$, and a naive query of

```
?- caused(q, V, 1).
```

(assuming an appropriate initial state and narrative) might compute the answer $V \mapsto \mathtt{t}$ three times. In the presence of long narratives and complicated action descriptions of $\mathcal{EC}+$, this recomputation of identical answers can ramify to an intolerable degree. In cases where variables occur in the constant, rather than the value, this is particularly undesirable, for—in an example schematic but all too realistically illustrative—one might wait for scores of identical answers $X \mapsto$ *monkey*, $V \mapsto$ *onBox* to a query of `caused(loc(X),V,30)` before moving on to the other answer $X \mapsto$ *bananas*, $V \mapsto$ *onTree*.

As the proof procedure recurses down through the clauses of our logic program, moving, by the axioms given in Section 3.2.4, from later times to earlier,

variables present in the top-level query become gradually more ground. We will record the answer substitutions for our top-level query as facts in the dynamic PROLOG database, and will adapt our logic programs, so that a list of the variables present in our top-level query is carried around the search tree. Whenever the top-level variables become instantiated in ways which we have already seen, we automatically fail, thus removing a potentially enormous amount of wasted computation in the search tree. This checking to make sure that top-level groundings are not repeated can be done at all depths of the recursive proof-procedure.

Clearly it would be possible to be even more thorough-going in checking for repeated answer substitutions. For suppose that a Boolean action description contained the causal laws

$$\textbf{inertial } p(X), q(X), r(X,Y)$$
$$\textbf{caused } p(X) \textbf{ if } \top \textbf{ after } q(Y) \wedge r(X,Y).$$

If our top-level query is `caused(p(X),V,30)`, then checking for identical groundings of the variable X would not prevent, when the system tries to prove `caused(q(Y),tt,29)`, identical substitutions for Y being found. Yet keeping track of all variables at all levels of the search tree, in the way that this suggests, quickly becomes costly, and we have chosen to implement the verification only for variables in the top-level query. (Compromises of looking at the first n different variables, or variables in the first n levels of recursion, are easily imaginable.)

The tabling of computed answers is another way we avoid unnecessary recomputation in our logic programs. This solves a different (if related) problem to that which checking for repeated answer substitutions addresses. In tabling, we record answers to queries of `caused/3`, at all levels of recursion, everywhere in the SLDNF search tree. A query which succeeds to give a grounding of, say, `caused(c,v,30)`, is retained for the duration of the session, so that it is available to subsequent top-level queries. (This is unlike the case with variable substitutions which have been seen, information about which must be specific to top-level queries, and which is abolished from the dynamic database between top-level queries.) Alongside information about the specific ground atoms which have been proved, we also record information about the form of the query which was ground to produce the answer. Thus if *all* answers to a query (possibly containing variables) were found and tabled, we record the fact that the complete answer substitutions to the query are stored in the database: we know that in this case, the axioms need not be used in answering the query.

So, when the system needs to prove some atom `caused(c,v,t)`, where c and v may have variables, it first checks to see whether a query like this, or one more general than it, has been made before, where all answers were found and tabled. If this was so, the system will ignore the axioms, and simply extract the answers from the database. If such a completely answered query was not made, there may still be tabled answers in the database: our logic program retrieves any of these which may exist, then proceeds to using the axioms to try and prove any remaining answer substitutions.

And at the same time as this process of reading and writing to the table of answers is taking place, the system is still checking to make sure that duplicated substitutions of the top-level variables are not being made. It can be seen

that the two processes are not performing the same work, for first, as has been mentioned, the tabled values are retained between top-level queries, whilst information about variable substitutions is, necessarily, lost between these queries. Second, it may happen that at different nodes of the search tree, essentially the same query needs to be proved, although the variables involved are unrelated. So, suppose that some way into a computation whose top-level query is `caused(p(X),V,30)`, we are required to answer a query `caused(q(Y),V',25)`. Say the search tree is as in Figure 3.3. We have simplified the representa-

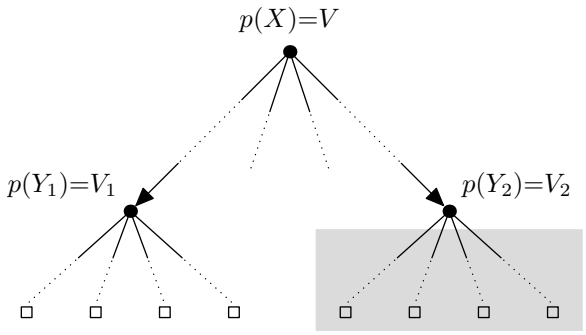

Figure 3.3: Sample search tree; the shaded sub-tree is redundant.

tion in inessential ways, by removing time indices and the outer casing of the 'caused' predicate. After the subtrees beginning at all four children of the left-hand node have been traversed, the PROLOG database contains all answers for a query `caused(p(Y1),V1,t)` (where t is the relevant time). Now, even if the system recorded information about *all* variable groundings occurring in computed answers (and not just the top-level ones), checking against the groundings which have been already seen would not help us, if we presume that Y_1 and Y_2 are unrelated (e.g. that they are not both bound to X). Yet the right-hand node in the search tree is clearly redundant, as it represents the same query as the left-hand, and modulo the name of the variable concerned, is the root of an identical sub-tree. Tabling will remove the necessity of re-traversing the whole sub-tree: in our modified implementation for $\mathcal{EC}+$, the answers of the right-hand sub-tree would be extracted immediately from the tabled data. During the process of developing $\mathcal{EC}+$, it was found repeatedly that repetitions of this sort were ubiquitous, and that their removal through the methods of tabulation greatly improved the speed of our computations.

In addition to the two methods of checking for duplicated variable substitutions and tabling the results of computations, we partially evaluate our logic programs, and of course avail ourselves of the features of PROLOG's indexing method for predicates. We do not comment on that process here, but move on now to present details of the modifications made to our logic programs in order to perform tabling and variable-checking. As stressed, the implementation of the tabling owes its inspiration to [dSA95].

3.7.1 Information Stored

Suppose our query (which may or may not be top-level) is for *caused*$(C,\ V,\ T)$. C may contain unbound variables, and V may be or contain an unbound variable. Let C have the form *cfunc*$(\vec{x})$, where the tuple $\vec{x}$ is possibly of zero length—in that case C is identical to *cfunc*. It may be that there was previously a more general query than the current, for which the system knows it stored all the answers. If this is so, then there will be a fact

$$calls\_cfunc(t,\ cfunc(\vec{y}),\ V') \tag{3.3}$$

in the database such that for some θ, $\vec{y}\theta = \vec{x}$ and $V'\theta = V$. θ can be the identity substitution. In general, we will call atoms of the form (3.3) *callterms*.

If callterms store the questions asked, *tableterms* store the answers. Let us suppose our query above had an answer, with the substitution θ'. Then the following atom will be a fact in the PROLOG database:

$$table\_cfunc(t,\ C\theta',\ V\theta').$$

Our program returns ground answers to its queries, and thus tableterms which are tabled as facts in the database will always be ground; this need not be so for callterms.

Finally, we turn from tabled questions and answers to stored variable instantiations. Suppose our top-level query is for *caused*$(C,\ V,\ T)$ as before, and let the free variables in C and V be collected into a list which is bound to `List`—this can be achieved by a call to the PROLOG in-built predicate `free_variables/2`, of the form

$$free\_variables(dummy(C,\ V),\ List)$$

Each unique answer to our query corresponds to a different instantiation θ of the list of variables *List*. Accordingly, whenever we succeed with our top-level query, we store a fact

$$var\_seen(List\theta)$$

in the database. As has been explained, one can check the current state of the top-level variables *List* during the progress of a computation, to ensure one will not duplicate a result.

In summary, we store dynamically facts of the following predicates:

- *calls_cfunc*/3, for *cfunc*$(\vec{x}) \in \sigma^f$, with $\vec{x}$ possibly empty;
- *table_cfunc*/3, similarly;
- *var_seen*/1, with the argument a list of ground terms.

3.7.2 `assert_callterm/5`

In order to enable easy generation of callterms and tableterms, we construct templates for these when action descriptions are compiled into memory. For each *cfunc* and each n such that *cfunc*$(\vec{x}) \in \sigma^f$ with $\vec{x}$ being an n-tuple, we include facts in our database:

$$callterm(cfunc(\vec{y}),\ V,\ T,\ calls\_cfunc(T,\ cfunc(\vec{y}),\ V))$$

and

$$tableterm(cfunc(\vec{y}),\ V,\ T,\ table\_cfunc(T, cfunc(\vec{y}),\ V)).$$

Here, $\vec{y}$ is an n-tuple of new, unbound variables.

We also make use of the predicate `assert_callterm/5`, which will be used when we need to update the callterms in our database, after all answers to a given query have been found. Here is the clause which defines this predicate:

```
% ---------- assert_callterm/5 ----------

assert_callterm(C, V, T, N, Flag) :-
    callterm(C, V, T, CallTerm),
    (
        N = 0,
        copy_term(CallTerm, CCallTerm),
        CallTerm,
        subsumes(CCallTerm, CallTerm),
        retract(CallTerm),
        fail
        ;
        (
            (N = 0 ; Flag = 1)
            ->
                assert(CallTerm)
            ;
                true
        )
    ).
```

After we have extracted relevant answers for a query from the database, and any remaining answers have been proved using methods of the original `caused/3`, we call `assert_callterm/5`. Let our query have been for $cfunc(\vec{x}) = V$ at time t. We will have the following bindings on calling the predicate:

`C` $\mapsto$	$cfunc(\vec{x})$
`V` $\mapsto$	V
`T` $\mapsto$	t
`N` $\mapsto$	0 if the query is top-level
	1 if the query is not top-level
`Flag` $\mapsto$	1 if the query had no free variables, we proved and tabled it
	0 otherwise

Note that $\vec{x}$ and V are not themselves bound (at this point) in accordance with any of the answers we may have proved along the way, but are as they were when the query was originally posed.

The last two arguments of this predicate deserve some comment. If the query is top-level ($N = 0$), then we can be assured that, when `assert_callterm/5` is called, all answers to the query have been found and tabled. This may not

be the case if the query is not top-level ($N = 1$): for in that case, answers to a query may be prevented from being found by the checks on duplicate variable instantiations. However, we *can* be assured that all answers to a lower-level query have been found and tabled when that query contained no free variables and one answer was found ($Flag = 1$).

So, stepping through the workings of the predicate: if the query is top-level ($N = 0$), we can delete all the callterms in the table which are more fully instantiated than the query. To do this we remove callterms from the database and test to see whether they θ-subsume the callterm of the query; if they do, we retract them. That is the work of the lines

```
copy_term(CallTerm, CCallTerm),
CallTerm,
subsumes(CCallTerm, CallTerm),
retract(CallTerm),
fail
```

which remove subsumed callterms until there are none left. Clearly, if the query was not top-level, but was fully ground, then it can have no callterm in the database, for otherwise (given the definition of `caused/5` below) the predicate `assert_callterm/5` would not have been called. When all outmoded callterms have been removed, we may go on to assert a new callterm based on the current, recently-completed query. We may only do this when we can guarantee that all answers for the query have been produced, and as the discussion above shows, these circumstances hold when the test

```
(N = 0 ; Flag = 1)
```

succeeds.

3.7.3 caused/5

We must now modify the definition of `caused/3` in order to accommodate procedures for tabling and the checking of duplicate variable groundings. Extra arguments will be added, corresponding to the variable N which carries information about whether the query is top-level or not, and the list of variables *Vars* for the top-level query—which we will check against facts *var_seen*/1 in our database. The essence of the code which was originally contained in the definition given earlier for `caused/3` will now be put in a predicate `caused_axiom/4`, which appears in the second of our new clauses.

```
% ---------- caused/5 ----------

caused(C, V, T, _, Vars) :-
    callterm(C, V, T, CallTerm),
    copy_term(CallTerm, CCallTerm),
    CallTerm,
    CallTerm =@= CCallTerm,
    !,
    tableterm(C, V, T, TableTerm),
    TableTerm,
    \+ (var_seen(Seen), Seen == Vars).
```

```
caused(C, V, T, N, Vars) :-
    copy_term(C, CCopy),
    copy_term(V, VCopy),
    tableterm(C, V, T, TableTerm),
    (
        (
            TableTerm
            ;
            caused_axiom(C, V, T, Vars),
            (
                TableTerm
                ->
                    true
                ;
                    assert(TableTerm)
            )
        ),
        (
            C == CCopy
            ->
                !,
                assert_callterm(C, VCopy, T, N, 1)
            ;
                true
        ),
        \+ (var_seen(Seen), Seen == Vars),
        (
            N = 0
            ->
                ensure_asserted(var_seen(VC, Vars))
            ;
                true
        ),
        ;
        assert_callterm(C, V, T, N, 0),
        fail
    ).
```

We make use of common PROLOG predicates, and also the less common `=@=/2`, which tests for 'structural equivalence'. This is stronger than unification but weaker than `==/2`. `T1 =@= T2` is true when their tree representation is identical and they have the same pattern of variables, so that

```
a(X) =@= a(Y),   a(X, Y) =@= a(Y, Z)
```

are true but

```
a(X) =@= a(a),   a(X, Y) =@= a(Z, Z)
```

are not.

Suppose our query is for C having value V at time t, and let C have the form $cfunc(\vec{x})$ as usual. We enter the first clause defining `caused/5` and make a callterm for our query. We then check to see whether there is a more (or equally as) general callterm in the database. If there is, then we cut: we know that we can get all answers for the current query by looking in the table, and so there is no need to continue to the second clause defining `caused/5`, which we use when we need the details of `caused_axiom/4` to prove answers. After the cut, if we get that far, we make a tableterm and extract answers from the table—obtaining multiple answers by backtracking.

What of the line

```
\+ (var_seen(Seen), Seen == Vars) ?
```

If our query is top-level, then there will be no `var_seen/1` facts in the database, and so this query will succeed. If, on the other hand, the query is not top-level, then we will not want the query to succeed if it has instantiated variables in *Vars* in a way we already seen before, for a computed answer to our top-level query.

Now for the second clause. This will only have been reached if there is no more general callterm in the table. There may however still be relevant tableterms in the table, so after making copies of the details of our original theory (which will be used later), we fetch a tableterm and get values from the table. If we run out of answers stored in the table, we use our axioms to prove new values, tabling the latter as we progress.

As has been stated, though our queries may contain variables, when answers are given all those variables must be fully instantiated. Therefore, if after we prove our query the C component is the same as it was before the proof, then that C before proof must have been ground. In that case, there is no need to backtrack for further answers to our query, and we can cut. That is the purpose of the next stage:

```
(
    C == CCopy
    ->
        !,
        assert_callterm(C, VCopy, T, N, 1)
    ;
        true
),
```

Note that if we *do* cut at this stage, we also update the callterms in our database; we call `assert_callterm` with, importantly, a 1 as its last parameter—denoting the fact that the query has no variables.

If a cut was not made after the `C == CCopy` test as described above, we record a *var_seen*/1 fact if necessary. Backtracking may then show different values for our query (it is only guaranteed to show *all* such values when the query is top-level, because of the presence of checking against seen variables). After we have backtracked as much as possible, the following is reached:

```
assert_callterm(C, V, T, N, 0),
fail
```

The operation of this predicate was described in Section 3.7.2.

3.7.4 Axioms

The definition of `caused_axiom/4`, and related predicates, should be easily understood, as it builds closely on the original definition for `caused/3`. The subsidiary predicates given along with that original definition have been modified.

```
--------------- caused_axiom/4 ---------------

caused_axiom(C, V, 0, _) :-
    init(C, V).

caused_axiom(C, V, T1, Vars) :-
    0 < T1,
    causes(C, V, A, H),
    T is T1 - 1,
    all_happen(A, T),
    all_caused(H, T, Vars).

caused_axiom(C, V, T, Vars) :-
    0 < T,
    causes(C, V, F),
    all_caused(F, T, Vars).

caused_axiom(C, V, T1, Vars) :-
    0 < T1,
    inertial(C, V),
    T is T1 - 1,
    caused(C, V, T, Vars),
    \+ clipped(C, V, T, T1).

caused_axiom(C, V, T, _) :-
    0 < T,
    default(C, V),
    \+ overridden(C, V, T).

% --------------- all_happen/2 ---------------

all_happen([], _).

all_happen([C=V|Rest], T) :-
    happens(C, V, T),
    all_happen(Rest, T).

% --------------- all_caused/3 ---------------

all_caused([], _, _).

all_caused([C=V|Rest], T, Vars) :-
    caused(C, V, T, 1, Vars),
    all_caused(Rest, T, Vars).
```

```
% --------------- clipped/4 ---------------

clipped(C, V, T1, T2) :-
    domain(C, V1),
    V1 \= V,
    tableterm(C, V, T1, TableTerm),
    (
        TableTerm
        ->
            true
        ;
            (
                causes(C, V1, A, H),
                all_happen(A, T1),
                all_caused(H, T1, [])
                ;
                causes(C, V1, F),
                all_caused(F, T2, [])
            ),
            callterm(C, V1, T, CallTerm),
            (
                CallTerm
                ->
                    true
                ;
                    assert(CallTerm)
            ),
            assert(TableTerm)
    ).

% --------------- overridden/3 ---------------

overridden(C, V, T) :-
    domain(C, V1),
    V1 \= V,
    caused(C, V1, T, 1, []).
```

That ought to be straightforward, apart, possibly, from the definitions of the final two predicates, `clipped/4` and `overridden/3`. In particular, one might wonder why the *Vars* parameter has been changed to the empty list `[]` in these predicates, the original *Vars* not being passed down from where they are called.

First, let us ask what it would mean if we come to a call of `clipped/4` or `overridden/3` in a state where a fact *var_seen*([]) has already been asserted in our database. This would be true if our top-level query had no variables in at all—the list of variables seen has no members—and if we had already verified the query. Yet then clearly in that case the test `C == CCopy` would have been passed, and the ensuing cut would have excised the possibility of another call to `caused_axiom/4`. In other words, it is *impossible* to come to a call of `clipped/4`

or `overridden/3` in a state where a fact *var_seen*([]) has already been asserted. Therefore any checks of the form

```
\+ (var_seen(Seen), Seen == Vars)
```

we perform *beneath* those calls will succeed. The purpose of ignoring the current binding of *Vars* on entering such calls, is therefore to 'ignore', in a sense, the checks for seen variables.[6] Why should we want to ignore these tests? The answer is that because the predicates `clipped/4` and `overridden/3` occur within the scope of a negation-by-failure (NBF) operator, any bindings occurring in the new SLDNF tree growing beneath those calls will be ignored if the NBF call succeeds. Thus such bindings cannot contribute to a further instantiation of the variables *Vars* of our top-level query.

3.8 Comparison of Implementations

There are now two different implementations available for answering queries about narratives defined in $\mathcal{EC}+$. Since action descriptions of $\mathcal{EC}+$ are also sets of laws of $\mathcal{C}+$, one way of answering queries is to submit the laws, together with information about the initial state and subsequent actions, to CCalc, and ask CCalc to find models of Γ_t^D which are consistent with the input narrative. Another way is to use the logic-programmed implementation of $\mathcal{EC}+$ which we presented in Section 3.2. As the objective of this work has been to make the answering of queries about fluents much more efficient, by using an event-style of computation, it is appropriate to compare the performance of our logic programs to that of CCalc on some sample domains.

We will first consider our running example, the 'farmyard' domain of Section 2.1.7. The signature of that domain has

$$\sigma^f = \{alive(x), loaded, smiling(x), target, loc(x)\},$$
$$\sigma^a = \{aim, miracle(x), load, shoot, walk(x)\},$$

for $x \in \{Bill, Turkey\}$. Domains are Boolean, except that

$$\begin{aligned} dom(loc) &= \{barn, field, house\} \\ dom(target) &= \{barn, field, house, none\} \\ dom(aim) = dom(walk) &= \{barn, field, house, none\}. \end{aligned}$$

The only static laws in the action description are the two relating the fluent constants *alive*(x) and *smiling*(x); together they mean that, of the four combinations of these fluents for each grounding of x, only three can occur:

$$\{alive(x), smiling(x)\},\ \{alive(x), \neg smiling(x)\},\ \{\neg alive(x), \neg smiling(x)\}$$

There are, in the light of this constraint and the absence of any others, 648 states in the transition system which the action description defines. Construction of the system by that component of our implementation which writes a representation of the transition system to an external file (see Section 3.5) confirms this, and also shows that there are 133068 transitions; these results are independently

[6] It would evidently be possible to use a different 'dummy' argument than the empty list.

confirmed by CCalc with `mchaff` as its propositional satisfaction solver (using `grasp` gives incorrect results).

For our first experiment we used the farmyard domain as it stands, with very simple, related runs, of increasing length. The runs are as represented in Figure 3.4. In the diagram, the initial state is twice-circled. In all states of

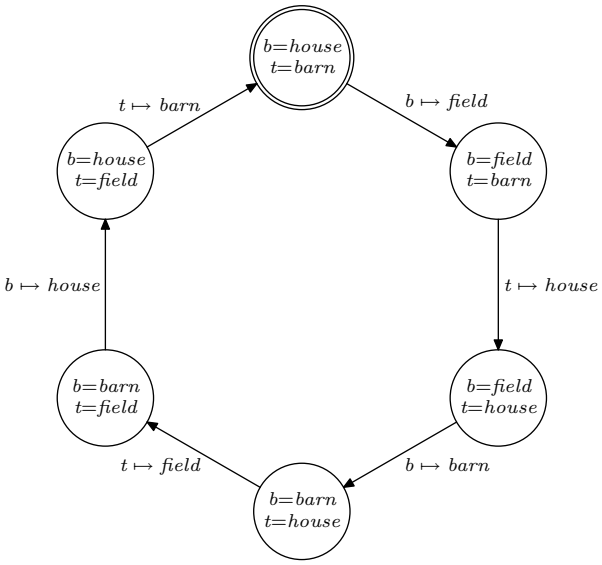

Figure 3.4: Runs for the farmyard.

the runs we have ¬*loaded*, *target*=*none*, *alive*(*bill*), *alive*(*turkey*), ¬*smiling*(*bill*), *smiling*(*turkey*). *b*=*house* represents *loc*(*bill*)=*house*, and $b \mapsto$ *field* represents *walk*(*bill*)=*field*. No miracles, aiming, loading or shooting happen on any edges; where walking isn't mentioned, it doesn't occur. (We can think of the run as representing *Bill* chasing the *Turkey* around the farm.)

We started with a run of length 500, and increased the length of the run in steps of 500, up to 6000 time-steps. Complete information about the initial state, and complete details of the narrative of actions, were given to our implementation of $\mathcal{EC}+$ and to *CC*Calc. For each length of run ($t = 500, t = 1000, \ldots, t = 6000$) we asked $\mathcal{EC}+$ to narrate the entire history five times, and recorded the time taken for computation, ignoring time spent on printing information about the run to the terminal. The results were averaged. We then performed the same experiment with CCalc. The results are shown in Table 3.1, and have been depicted graphically in Figure 3.5. Experiments were performed using SICStus Prolog 3.12.2 under Linux, with a Pentium IV 3.2GHz processor and 1GB RAM. The results are encouraging, in that, even when constructing the entire run through the transition system, $\mathcal{EC}+$ significantly outperforms CCalc.

The second experiment used a similar action description, again based on the farmyard domain. This time, we wished to hold the length of run constant (we used a run length of 20) and investigate what happened when the number of fluents in the signature, and hence laws in the action description, was increased. The action descriptions and signatures are identical to that presented in Section 2.1.7, where we first described the farmyard domain, except that the

Length of run	Average time (seconds)	
	CCALC	$\mathcal{EC}+$
500	8.406	1.592
1000	28.964	5.556
1500	55.376	11.908
2000	93.508	21.808
2500	143.622	31.728
3000	201.034	45.332
3500	274.902	61.974
4000	351.212	80.328
4500	442.600	99.710
5000	541.580	123.756
5500	661.260	148.442
6000	781.026	175.958

Table 3.1: Comparison of CCALC and $\mathcal{EC}+$, `narrate` query.

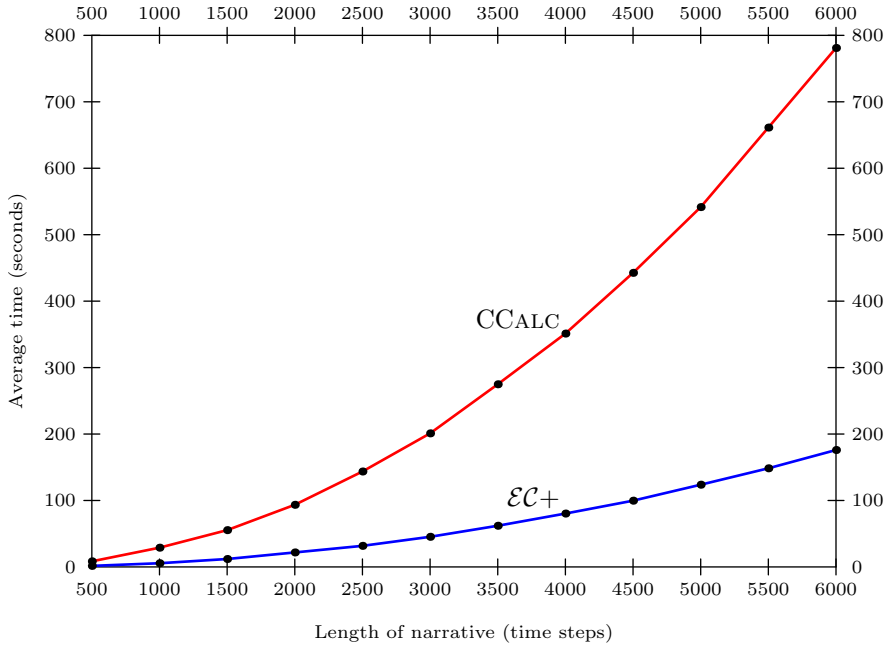

Figure 3.5: Computation times for the farmyard runaround.

variable x, which originally ranged over the set

$$\{Bill, Turkey\},$$

will now range over

$$\{Bill(i), Turkey(i) \mid 1 \leqslant i \leqslant n\}.$$

A series of action descriptions was obtained by taking values of n from the set $\{20, 40, \ldots, 200\}$, and the same queries were run, and averaged as before.

The results are shown in Table 3.2 and Figure 3.6. Again, $\mathcal{EC}+$ shows good

Value of n	Average time (seconds)	
	CCALC	$\mathcal{EC}+$
20	6.454	0.52
40	21.106	2.672
60	44.172	7.498
80	76.448	15.862
100	115.286	28.666
120	164.852	46.378
140	221.666	71.824
160	288.492	103.136
180	360.152	143.112
200	445.616	193.274

Table 3.2: Comparison of CCALC and $\mathcal{EC}+$, `narrate` query on the 'busy farmyard'.

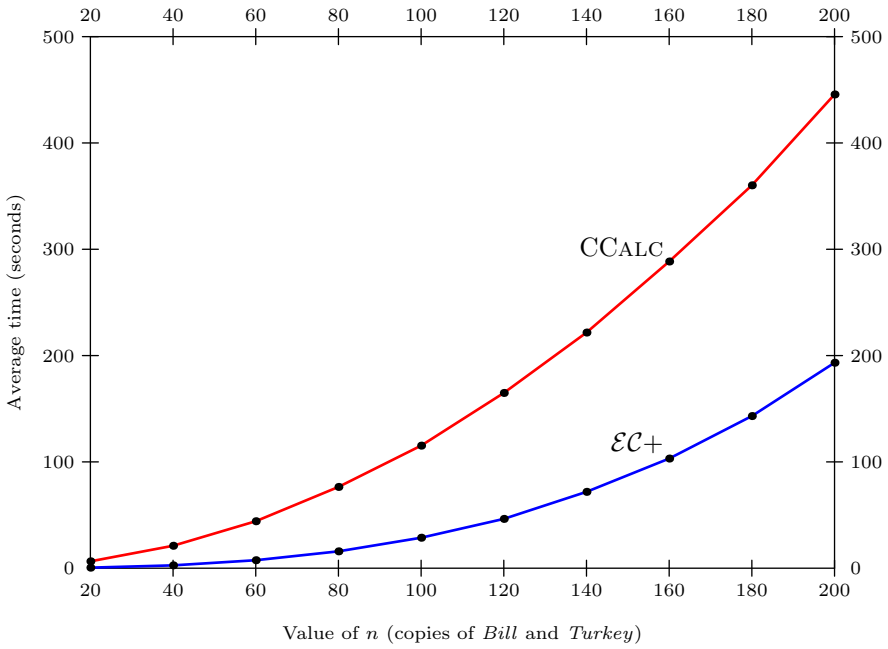

Figure 3.6: Computation times for the busy farmyard.

performance compared to CCALC, though in this case the way of making the domain more complicated (by increasing the number of fluents and laws in the action description) appears to make the advantage of using $\mathcal{EC}+$ over CCALC of a lower order.

How does $\mathcal{EC}+$ fare when we turn from constructing the entire run through a transition system, to more specific queries? The answer to this, naturally, depends on the nature of the query and the laws of the action description which

defines the transition system. For example, given the narrative of actions for the farmyard domain which is shown in Section 3.6, a query asking for the value of *loc*(*Bill*) at time 12 would need to look back only as far as the actions occurring at time 10, when *walk*(*Bill*)=*house* was true. The fact that no more distant history needs to be considered is a consequence of the presence of the law

$$walk(x)=l \textbf{ causes } loc(x)=l$$

in the action description, the inertia of $loc(x)$ fluents, and the absence between times 10 and 12 of any actions which would disturb the value of *Bill*'s location. Clearly, if a narrative of events for this action description has a form similar to

$$(s_0, e_0, s_1, e_1, \ldots, e_{t-1}, s_t)$$

where $e_{t-1} \models walk(x)=l$, and we are interested in the value of l' for which $s_t \models loc(x)=l'$ holds, then $\mathcal{EC}+$ should able to provide us with an answer extremely quickly, regardless of the value of t, whereas CCalc will take longer as we increase t.

Experiments conducted with the series of runs we used previously in the 'farmyard runaround' example confirmed this: in each case we asked, given a run of length t, what the value of *loc*(*bill*) was at time t. In $\mathcal{EC}+$ this was achieved by a query of the form

```
q(loc(bill)=V, t),
```

and the results were, in almost all cases, a time of 0 milliseconds. In CCalc we extracted the answer by posing a query which included information about the initial state and events at all times of the run; these are, of course, identical to the queries we made in the first experiment of this section, and thus the times can be read from Table 3.1.

This query represents cases which are easy for $\mathcal{EC}+$, demonstrating a phenomenon which occurs frequently: that increasing the length of run does not effect the time taken to answer specific queries. The same is often true when we increase the number of fluents in the signature, as we did with the experiments on 'busy farmyard' domains, shown in Table 3.2 and Figure 3.6; answers to a query the location of *Bill1*, in a domain with millions of copies of *Bill*, would be just as quickly answered—though again, this is a consequence of the specific details of the action description and narrative.

3.9 The Zoo World

As a longer case study of the expressive possibilities of $\mathcal{EC}+$ and the various forms of efficiency it affords, we include a formalization of the 'Zoo World', a common test domain for reasoning about action and change.

The Zoo World was first proposed by Erik Sandewall,[7] and has been formalized as a $\mathcal{C}+$ action description [AEL+04]; it is one of several sample domains which comes bundled with the system CCalc, and so it has been easily possible to conduct experiments with that domain and test the adequacy of its formulation in $\mathcal{C}+$. (Readers are expected to be familiar with the $\mathcal{C}+$ formalization

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

in the following.) We will not directly translate the $\mathcal{C}+$ action description—let us call it D_{zoo}—though we have been guided by that formulation in our interpretation of the original, loose specification for the domain.

The prose description of the domain mentions a *throwoff* action, which

> [c]an be performed by an animal ridden by a human, and results in the human no longer riding the animal and ending in a position adjacent to the animal's present position. The action is nondeterministic since the rider may end up in any such position. If the resultant position is occupied by another large animal then the human will result in riding that animal instead.[8]

The transition systems defined by action descriptions of $\mathcal{EC}+$ cannot be nondeterministic, and so we have chosen to eliminate *throwoff* from the actions which animals in the zoo may perform. Apart from this restriction, the behaviour we model is the same as in D_{zoo}.

Our formalization is shown, in full, in Appendix B. It is, as one would expect, very similar to the version in $\mathcal{C}+$; we discuss significant differences below.

One important difference is to be found in the treatment of properties of the domain which do not change over time. These are such facts as that there are 3 cages, that there is an animal called *Dwight* whose species is *Dingo*, that there is a gate whose two sides are the positions called p_1 and p_2, and so on. It would be possible to represent all of this by fluent constants, yet we have chosen to use PROLOG clauses, using the predicates which these clauses define to make specific what is, essentially, a parameterized action description and signature. For instance, consider a zoo of the form given in Figure 3.7. There

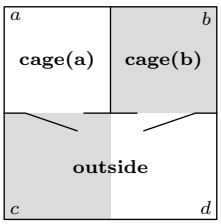

Figure 3.7: A sample Zoo topography

are four *positions*, namely a, b, c and d, and three locations, **cage(a)**, **cage(b)** and **outside**. Two gates, *gate*(a) and *gate*(b), connect the relevant cage to the outside. The relations between positions, locations and gates are as depicted in the diagram. Now, it would be possible to introduce Boolean fluent constants *neighbour*(p_1, p_2) for every position p_1 and p_2 in the domain, and to express the extension of the neighbour relation as causal laws in the action description, such

[8] Ibid.

as

$$\begin{aligned}
&\textbf{caused } \mathit{neighbour}(a,b)\text{=}\mathbf{t} \textbf{ if } \top,\\
&\textbf{caused } \mathit{neighbour}(b,a)\text{=}\mathbf{t} \textbf{ if } \top,\\
&\textbf{caused } \mathit{neighbour}(b,d)\text{=}\mathbf{t} \textbf{ if } \top,\\
&\textbf{caused } \mathit{neighbour}(d,b)\text{=}\mathbf{t} \textbf{ if } \top,\\
&\textbf{caused } \mathit{neighbour}(c,d)\text{=}\mathbf{t} \textbf{ if } \top,\\
&\textbf{caused } \mathit{neighbour}(d,c)\text{=}\mathbf{t} \textbf{ if } \top,
\end{aligned}$$

and so on, with the appropriate other laws for positions which are not neighbours. (Since we do not allow statically-determined fluent constants in $\mathcal{EC}+$, we cannot insist that $\mathit{neighbour}(p_1,p_2)$ is statically determined and then simply say

$$\textbf{default } \mathit{neighbour}(p_1,p_2)\text{=}\mathbf{f} \qquad (p_1,p_2 \in \{a,b,c,d\})$$

for here the exogeneity of the initial state would provide us with unintended models of Γ^D_t.) Yet we do not introduce fluent constants in this way. The relevant portion of our logic program is this:

```
% ---------- General

neighbour(P1,P2) :-
  neighbour(P1,P2,l).

neighbour(P1,P2,_) :-
  nb(P1,P2).

neighbour(P1,P2,_) :-
  sides(_,P1,P2).

neighbour(P1,P2,l) :-
  neighbour(P2,P1,r).

sides(gate(a),a,c).
sides(gate(b),b,d).

flu_constant(accessible(P1,P2)) :-
  neighbour(P1,P2).

% ---------- Specific

nb(c,d).
```

In this way, we parameterize the signature (in the current case, which fluent constants are of the form $\mathit{accessible}(p_1,p_2)$) on the *neighbour* relation which we have defined in the PROLOG background of our logic program. The 'general' clauses above are included in every Zoo World action description; all we need to include in each particular instantiation of that world's governing principles, is a record, in `nb/2`, of those positions not either side of a gate which are next to each other. In the $\mathcal{C}+$ action description of [AEL+04], there is a Boolean fluent constant $\mathit{accessible}(p_1,p_2)$ for every two (not necessarily distinct) positions p_1 and

p_2 in the world, something which, over long runs, clearly multiplies the number of clauses sent to the SAT-solver without good reason—whereas we include only those fluent constants $\mathit{accessible}(p_1, p_2)$ which could possibly be true, given the unchanging information about the structure of the zoo. This sort of parameterization, using the very efficient underlying PROLOG, is a clear advantage to the formulation of action domains in $\mathcal{EC}+$ and its logic programs. (Another advantage of our representation is that it encodes directly the constraint that the *neighbour* relation is symmetric.)

Many laws in the $\mathcal{C}+$ formulation of the Zoo World are constraints, causal laws with $\perp$ as their head, the effect of which is to remove states and transitions from the transition system the formulation defines. In checking that $\perp$ is never caused, which forms part of the process of ensuring that the input narrative and initial state define a run through the transition system defined by our action description (see Section 3.4 for more details) we must verify that the bodies of these constraints are never true of any state or transition. Now, some of the constraints make reference to properties of the domain which must be modelled as fluent constants, as they change over time. For instance, the specification states that an *Open Gate* action

> [c]an be performed by a human when it is located in a position to the side of the gate...[9]

and since positions change over time, and the position of a human counts towards the identity of a state (i.e. it must be represented using fluent constants rather than action constants), the law

$$\textbf{nonexecutable } \mathit{open}(H, G) \textbf{ if } \mathit{pos}(H){=}P$$

where P is a position not to one side of the gate G, cannot be checked to be 'safe' in the manner we require, apart from details of the initial state and narrative. We may indeed look at the narrative of events to see at which times an action $\mathit{open}(H, G)$ is performed, and that would require no computation beyond looking at this input narrative; yet to know whether the relevant $\mathit{pos}(H){=}P$ holds at the same time, we may need to do much computation, using the axioms for $\mathcal{EC}+$ and in the manner described in Section 3.4.

However, there are other constraints which do not make reference to changeable properties of states in this way, and by our decision to use fluent constants only to represent properties of states which can change—keeping such static features of the domain as the zoo's topology in the background as parameters of the signature—we have been enabled to check these constraints by simple PROLOG code, which does not draw upon the causal inference mechanism for $\mathcal{EC}+$. The constraints we can check quickly should perhaps be divided into two sorts: those which make reference to the narrative of events and those which do not. For example, in the first category is a constraint stemming from the stipulation that

> [...] two large animals can not pass through a gate at the same time (neither in the same direction nor opposite directions).[10]

[9] Ibid.
[10] Ibid.

Now, the case of the above for "opposite directions" *could* be rendered by a causal law something like

$$\textbf{nonexecutable } \mathit{move}(A_1){=}P_1 \wedge \mathit{move}(A_2){=}P_2 \textbf{ if } \mathit{side}_1(G){=}P_1 \wedge \mathit{side}_2(G){=}P_2 \wedge \mathit{large}(A_1) \wedge \mathit{large}(A_2) \tag{3.4}$$

supposing that we had fluent constants $\mathit{side}_1(G)$ and $\mathit{side}_2(G)$, and so on; the $\mathcal{C}+$ formulation of the Zoo World in [AEL$^+$04] does something similar. Yet suppose instead we define a predicate `constraint/0`, one of whose clauses is

```
constraint :-
  large_animal(A1),
  large_animal(A2),
  A1 @< A2,
  happens(move(A1),P1,T),
  happens(move(A2),P2,T),
  (sides(_,P1,P2) ; sides(_,P2,P1)).
```

(The meaning of the body should be obvious: `sides(g,p1,p2)` is true when a gate denoted by g has positions $p1$ and $p2$ as its two sides.) We can load the clauses defining `constraint/0` into PROLOG with the $\mathcal{EC}+$ logic program representing the Zoo World, and query `constraint`. If this succeeds, then we know that the input narrative, together with the clauses representing facts about the zoo's topology and occupants, is inconsistent in the sense of not defining a run through the transition system. It is also clear that if a query of `constraint` were to fail, this would demonstrate that the causal law (3.4) is, in an important sense, unnecessary. Of course, adding (3.4) to the action description changes the labelled transition system it defines; but where D is the action description of $\mathcal{EC}+$ with, and D^* the action description without this causal law, we have that the models of

$$\Gamma_t^D \cup \{F \Leftarrow \top \mid \mathit{Init}, \mathit{Haps}_0, \ldots \mathit{Haps}_{t-1}\}$$

must be the same as those of

$$\Gamma_t^{D^*} \cup \{F \Leftarrow \top \mid \mathit{Init}, \mathit{Haps}_0, \ldots \mathit{Haps}_{t-1}\},$$

where the sets *Init* and Haps_i encode the particular initial state and narrative against which we check `constraint`. In other words, with this particular initial state and narrative of events, the inclusion of the **nonexecutable** law above would be unnecessary, and would simply lead to unnecessary further checking for consistency. We can safely omit it.

We have made use of this device of defining a 'constraint' predicate with the Zoo World, and so a preliminary stage of checking for consistency will always be to make sure that these constraints are satisfied. Given the large number of constraints we include, it has been useful to use a predicate of arity 1 rather than 0, to enable us to include identifiers for each constraint and so easily see, when the narrative and action description are inconsistent, what the reason is for the failure. The entire file `zoo_constraints.pl` is included as part of Appendix B.

We remarked above on the division of the description of each particular Zoo World domain into general principles and specific facts, the latter including

information about the initial state and narrative. Let us introduce a sample filling-out of the Zoo World's general principles to illustrate how concise our representations can be. To that end, consider the zoo shown in Figure 3.8. There are two animals: *ahab* the human and *moby* the whale; both are adults.

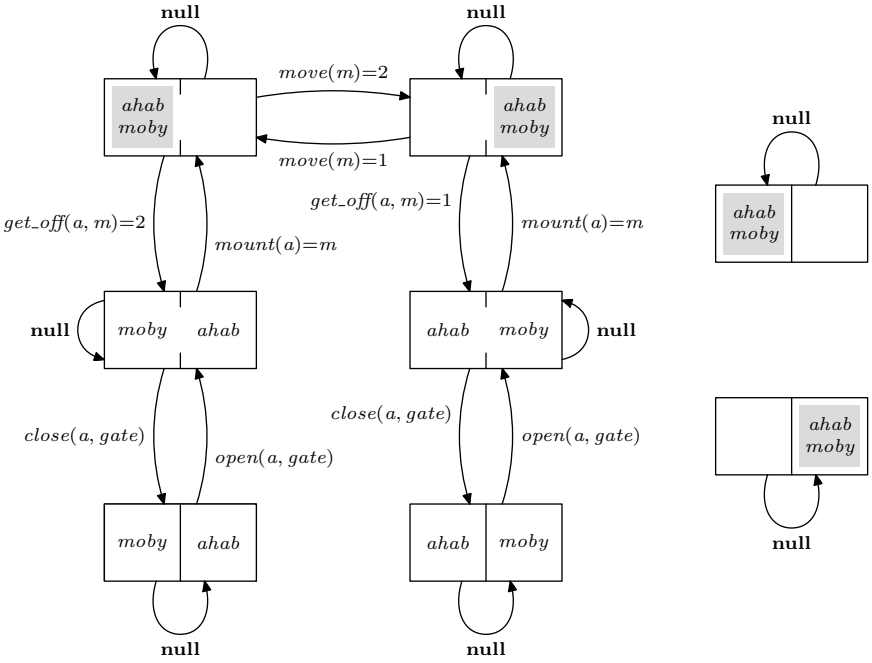

Figure 3.8: A small Zoo World

Humans and whales are large species. The zoo contains two positions, called 1 and 2; in the diagram position 1 is on the left, and 2 is on the right. The location of position 1 is the *cage*, and 2 is *outside*. There is a single gate, called *gate*. Where *ahab* and *moby* appear shaded in the same position, this indicates that *ahab* is mounted on *moby*. A **null** label on a transition indicates that all action constants are evaluated to **f** over that transition; on other transitions those action constants are evaluated to **f** which are not represented.

In conjunction with those causal laws and clauses which are held in common by all Zoo Worlds, the small domain depicted can be defined by the PROLOG clauses

```
gate(gate).
cage(cage).
sides(gate,1,2).
loc(1,cage).
loc(2,outside).
animal(ahab,human).
animal(moby,whale).
large_species(human).
large_species(whale).
```

We loaded our representation of this small zoo into our implementation for $\mathcal{EC}+$, and asked it to draw the labelled transition (functionality we mentioned in Section 3.5). The result was the system depicted in Figure 3.8.

Unfortunately we have not been able to compare the performance of the $\mathcal{EC}+$, logic-programmed representation of instantiations of the Zoo World with the $\mathcal{C}+$ version running with CCALC. Whilst the latter works for small examples satisfactorily, it will not run on larger examples. (The interface to some PROLOG library modules has been implemented incorrectly.)

3.10 Relation to the Event Calculus

We have said, rather loosely, that the logic program

$$LP(D, t, Init, \{Haps_i \mid 0 \leqslant i \leqslant t\})$$

represents an 'event-calculus' style of computation, or that it is 'inspired by' the event calculus, and readers familiar with any variant of the event calculus—such as the one given in Section 2.4—will see that this is true. Yet the nature of the correspondence between the event calculus and $\mathcal{EC}+$ remains unclear. In this section we will look at the details of the relationship. Since there exist many different variants of the event calculus, there is no unitary relationship to $\mathcal{EC}+$. Accordingly, we will choose the variant described in Section 2.4, and investigate its correspondence to $\mathcal{EC}+$.

Action descriptions of $\mathcal{C}+$ specify the laws according to which a system evolves; causal laws do not express properties of individual runs of the system, which is why $\mathcal{C}+$ needs to be supplemented by a query language whose semantics is founded on the labelled transition systems which action descriptions define. The event calculus programs $(Ax, E, Init, N, T)$ which we described in Section 2.4, however, do contain information about particular runs through the system: *Init* encodes properties of the initial state, and N is the narrative of events. Thus to relate $\mathcal{C}+$ and $\mathcal{EC}+$ to the event calculus programs we have defined, we will extract the laws of evolution from the event calculus programs.

Definition 3.11 Let σ be a multi-valued propositional signature, with the usual partitioning into fluent and action constants. Let Ax be axioms of the simplified event calculus axioms, partially grounded in ways respecting the signature σ, but where variables intended to represent time-points are left free. Let E be a set of clauses of the form

```
initiates(a=v', c=v, T) :-
  holds_at(c1=v1, T),
  ...
  holds_at(cn=vn, T).
```

Then (Ax, E) is an *event calculus system specification.* ⌟

Definition 3.12 Let σ be a multi-valued propositional signature ($\sigma = \sigma^f \cup \sigma^a$) and (Ax, E) be an event calculus system specification. The *action description*

corresponding to (Ax,E), written $D^{Ax,E}$, has signature σ and the causal laws **inertial** c, for all $c \in \sigma^f$; **exogenous** a, for all $a \in \sigma^a$; and

$$\textbf{caused } c{=}v \textbf{ if } \top \textbf{ after } a{=}v' \wedge c_1{=}v_1 \wedge \cdots \wedge c_n{=}v_n,$$

for all members of E of the form

```
initiates(a=v', c=v, T) :-
  holds_at(C1=V1, T),
  ...
  holds_at(Cn=Vn, T).
```

⌟

Observation 3.13 For any event calculus system specification (Ax, E), $D^{Ax,E}$ is an action description of $\mathcal{EC}+$. ⌟

Herbrand models, and hence stable models, of our event-calculus programs P will contain the following components:[11]

- the set *Init* of `initially/1` atoms;
- the narrative N of `happens/2` atoms;
- atoms of the predicate `holds_at/2`;
- atoms of the predicate `initiates/3`;
- atoms of the predicate `terminates/3`;
- atoms of the predicate `broken/3`.

We would like to be able to show that stable models of any event calculus program P founded on an event calculus system specification (Ax, E) correspond to stable models of the corresponding program of $\mathcal{EC}+$, as specified in Definition 3.12. Insisting that the initial states and narratives of simplified event calculus programs should be 'consistent' and 'complete' will assist us in showing this: the correspondence theorem we will prove would not hold in general, for initial states and narratives which may be *inconsistent* or *incomplete*, according to the sense we gave in Section 2.4.

In fact, it is easier first to show that stable models of (complete, consistent and acceptable) event calculus programs correspond to runs through the transition system defined by $D^{Ax,E}$; the way we do this is by using the property that runs of length m through the transition system defined by D are essentially models of Γ^D_m—Theorem 2.10. We can then use Theorem 3.10 (that relating models of the causal theories Γ^D_t to stable models of our logic programs for $\mathcal{EC}+$) to move to the desired result.

[11] We have not mentioned atoms of the predicates `</2`, `=</2` and `\=/2`: given the signature σ and length of narrative m of an event calculus program, the facts of those predicates which are included in the program are fixed, and we will tacitly assume that any stable model will include them.

Theorem 3.14 Let (Ax, E) be an event calculus system specification. Then for all event calculus programs $P = (Ax, E, Init, N, T)$ which are acceptable, complete and consistent, and for all interpretations X of the signature of P,

$$X \models_{\mathsf{C}} \Gamma_m^{D^{Ax,E}} \cup \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in Init\} \\ \cup \{a[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N\}$$

if and only if there is a stable model M of $(Ax \cup E \cup Init \cup N)$ such that

$$\texttt{holds\_at(c=v,t)} \in M \quad \text{iff} \quad X \models c[t]{=}v \\ \texttt{happens(a=v,t)} \in M \quad \text{iff} \quad X \models a[t]{=}v \tag{3.5}$$

Proof: By induction on the length m of narrative.

(Base case: $m = 0$.) For the 'only if' direction, let X be an interpretation of the signature σ_0 and assume

$$X \models_{\mathsf{C}} \Gamma_0^{D^{Ax,E}} \cup \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in Init\} \\ \cup \{c[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N\}. \tag{3.6}$$

N is empty for $m = 0$, so that as there are no static laws in $D^{Ax,E}$, and given the nature of the translation from $\mathcal{C}+$ to action descriptions to causal theories, (3.6) reduces to

$$X \models_{\mathsf{C}} \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in Init\} \\ \cup \{c[0]{=}v \Leftarrow c[0]{=}v \mid c \in \sigma^f, v \in dom(c)\}. \tag{3.7}$$

But then clearly X is just the interpretation of σ_0 which assigns v to c iff `initially(c=v)` $\in$ *Init*. We must show that there is a stable model M of $(Ax \cup E \cup Init \cup N)$ with `holds_at(c=v,0)` $\in M$ iff `initially(c=v)` $\in$ *Init*. This is clearly the case, essentially as no stable model of $(Ax \cup E \cup Init \cup N)$ can contain any `happens(a=v,i)` atom, thus there is no atom `broken(c=v,t1,t2)` in M. (The model M may contain `initiates/3` and `terminates/3` atoms if there are corresponding clauses in E and the bodies of those laws are true in M—in this case the variable representing time will be bound to 0. A similar phenomenon arises with the inductive step.)
For the 'if' direction, X again is an interpretation of σ_0, and let M be a stable model of $(Ax \cup E \cup Init \cup N)$, for which the relevant biconditionals (3.5) hold. Recall that the event calculus program must be complete and consistent in the senses defined in Section 2.4. It is easy to see that

$$X \models_{\mathsf{C}} \Gamma_0^{D^{Ax,E}} \cup \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in Init\},$$

as M must obviously contain an atom `holds_at(c=v,0)` iff it contains an atom `initially(c=v)`.

(Inductive step: assume true for $m = k$, show for $m = k + 1$.) For the 'only if' direction, let X be an interpretation of σ_{k+1}, and assume

$$X \models_{\mathsf{C}} \Gamma_{k+1}^{D^{Ax,E}} \cup \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in Init\} \\ \cup \{a[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N\} \tag{3.8}$$

We must find a stable model M satisfying the relevant biconditionals. Let N^- be the set formed from N by removing all atoms of the form `happens(a=v,k)`. So, consider the causal theory

$$\Gamma_k^{D^{Ax,E}} \cup \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in \mathit{Init}\} \quad (3.9)$$
$$\{a[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N^-\}$$

Clearly

$$X - (\{(c[k+1], X(c[k+1])) \mid c \in \sigma^f\} \cup \{(a[k], X(a[k])) \mid a \in \sigma^a\})$$

is an interpretation of the signature σ_k of this theory (3.9); call this diminished interpretation X_0. By properties of causal theories and their relation to action descriptions, X_0 is a model (in the sense of $\models_{\mathsf{C}}$) of the theory (3.9). Thus by the inductive hypothesis there exists, corresponding to X_0, a stable model M_0 of $(Ax \cup E \cup \mathit{Init} \cup N^-)$. We have that

$$\begin{aligned} \texttt{holds\_at(c=v,t)} \in M_0 \quad &\text{iff} \quad X_0 \models c[t]{=}v, \\ \texttt{happens(a=v,t)} \in M_0 \quad &\text{iff} \quad X_0 \models a[t]{=}v. \end{aligned} \quad (3.10)$$

We extend M_0 to a model M of $((Ax \cup E \cup \mathit{Init} \cup N)$ having the desired property. So, define

$$\begin{aligned} M = M_0 \; &\cup \{\texttt{holds\_at(c=v,k+1)} \mid X \models c[k+1]{=}v\} \\ &\cup \{\texttt{happens(a=v,k)} \mid \texttt{happens(a=v,k)} \in N\} \\ &\cup \{\texttt{initiates(a=v',c=v,k+1)} \mid \text{there is an } \texttt{initiates/3} \text{ clause} \\ &\qquad \text{clause (as in Def. 3.11) in } E, \text{ and } X \models c_1[k+1]{=}v_1 \wedge \cdots \\ &\qquad \cdots \wedge c_n[k+1]{=}v_n\} \\ &\cup \{\texttt{terminates(a=v',c=v,k+1)} \mid \text{there is an } \texttt{initiates/3} \text{ clause} \\ &\qquad \text{clause (as in Def. 3.11) in } E \text{ whose head is} \\ &\qquad \texttt{initiates(a=v',c=v'',T)},\ X \models c_1[k+1]{=}v_1 \wedge \cdots \\ &\qquad \cdots \wedge c_n[k+1]{=}v_n,\ v'' \in \mathit{dom}(c),\ v'' \neq v'\} \\ &\cup \{\texttt{broken(c=v,t1,k)} \mid \text{there is an } \texttt{initiates/3} \text{ clause} \\ &\qquad \text{clause (as in Def. 3.11) in } E \text{ whose head is} \\ &\qquad \texttt{initiates(a=v',c=v'',T)},\ X \models c_1[k]{=}v_1 \wedge \cdots \\ &\qquad \cdots \wedge c_n[k]{=}v_n,\ v'' \in \mathit{dom}(c),\ v'' \neq v', \\ &\qquad \texttt{happens(a=v',k)} \in N, \text{ for all } (0 \leqslant t_1 \leqslant k)\} \end{aligned}$$

M extends the narrative of M_0 in ways respecting the laws which govern the dynamic behaviour of the system, and adds appropriate atoms of `initiates/3`, `terminates/3` and `broken/3`. As M clearly has the properties (3.5), it remains to show that M is a stable model of $(Ax \cup E \cup \mathit{Init} \cup N)$. That proceeds by a simple, if lengthy, case analysis.

For the 'if' direction, let X be an interpretation of σ_{k+1}, and assume that M is a stable model of $(Ax \cup E \cup \mathit{Init} \cup N)$, such that the biconditional properties obtain. We must show that statement (3.8) holds. So, restrict M to a Herbrand model M_0 by removing all atoms of the form `holds_at(c=v,k+1)`, `happens(a=v,k)`, `initiates(a=v',c=v,k+1)`, `terminates(a=v',c=v,k+1)`, and also the atoms

`broken(c=v,t,k)` (for any t), from M. Then, using the same notation N^- as for the 'only if' part, M_0 is a stable model of $(Ax, E, Init, N^-)$, and so X_0 (defined as previously), which is an interpretation of σ_k, is a model of

$$\begin{aligned}\Gamma_k^{D^{Ax,E}} \; & \cup \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in Init\} \\ & \cup \{a[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N^-\};\end{aligned}$$

henceforth we will call this causal theory Γ_k. Now, the causal rules which are added when moving from Γ_k to

$$\begin{aligned}\Gamma_{k+1}^{D^{Ax,E}} \; \cup \; & \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in Init\} \\ \cup \; & \{a[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N\}\end{aligned}$$

(we will abbreviate this as Γ) are the following:

- $c[k+1]{=}v \Leftarrow c[k+1]{=}v \wedge c[k]{=}v$, for $c \in \sigma^f$, $v \in dom(c)$ (these express inertia);
- $a[k]{=}v \Leftarrow a[k]{=}v$, for all $a \in \sigma^a$, $v \in dom(a)$ (these express the exogeneity of actions at the last time step);
- $c[k+1]{=}v \Leftarrow a[k]{=}v' \wedge c_1[k]{=}v_1 \wedge \cdots \wedge c_n[k]{=}v_n$ such that there is an `initiates/3` clause of the usual form in E;
- $a[k]{=}v \Leftarrow \top$ for all `happens(a=v,k)` $\in N$.

Now, we will consider the members of $\sigma_{k+1} - \sigma_k$ in turn, showing that for each constant c in that set there is a unique atom present in the reduct $(\Gamma)^X$ (recall Section 2.1.3).
So, first suppose $c \in \sigma^f$. There are two possibilities: either the value of c changes between times k and $k+1$, or it remains the same. Thus if `holds_at(c=v,k)` $\in M$ and `holds_at(c=v',k+1)` $\in M$ for some $v' \neq v$, then this must be as a consequence of the presence of some clause

```
initiates(a=v'',c=v', T) :-
  holds_at(c1=v1, T),...,holds_at(cn=vn, T).
```

in E, with `holds_at(c1=v1,k)`$,\ldots,$`holds_at(cn=vn,k)` $\in M_0 \subseteq M$ and also `happens(a=v'',k)` $\in M$. But then $X \models c_1[k]{=}v_1 \wedge \cdots \wedge c_n[k]{=}v_n$ and $X \models a[k]{=}v''$ so that as

$$c[k+1]{=}v' \Leftarrow a[k]{=}v'' \wedge c_1[k]{=}v_1 \wedge \cdots \wedge c_n[k]{=}v_n,$$

we have $c[k+1]{=}v' \in (\Gamma)^X$. There can be no other $c[k+1]{=}v^* \in (\Gamma)^X$: certainly not from the 'inertial' rules, but neither any rule of the form

$$c[k+1]{=}v^* \Leftarrow a'[k]{=}v''' \wedge c'_1[k]{=}v'_1 \wedge \cdots \wedge c'_n[k]{=}v'_n$$

whose body is satisfied by X. For if there were such a rule, then a corresponding atom `initiates(a'=v''',c=v*,k)` would be in M, whose presence would also require an atom `broken(c=v',0,k+1)` (the second argument has been chosen arbitrarily; it makes no difference), and thus given the structure of the axiom describing the effects of actions in event calculus programs, we could not have

`holds_at(c=v',k+1)` $\in M$.
Thus, to recap: if `holds_at(c=v,k)` $\in M$ and `holds_at(c=v',k+1)` $\in M$ for some $v' \neq v$, then there is precisely one atom whose constant is $c[k+1]$ in $(\Gamma)^X$, and that atom is $c[k+1]{=}v'$.
Suppose alternately that the value of c remains the same as the system passes to time $k+1$. In that case, we have `holds_at(c=v,k)`, `holds_at(c=v,k+1)` $\in M$, then an analysis similar to that above shows that there is a unique atom in $(\Gamma)^X$ whose constant is $c[k+1]$, and that the atom in question is $c[k+1]{=}v$.
Finally, if $a \in \sigma^a$, then as N is consistent and complete, there is precisely one rule in

$$\{a[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N\}$$

with $a[k]$ in its head. As the only other rules in Γ with an atom whose constant is $a[k]$ as their head also have that atom as their body (the rules are those stemming from the universal exogeniety of action in our action descriptions), then for each constant $a[k]$ with $a \in \sigma^a$, $(\Gamma)^X$ contains precisely one atom: that $a[k]{=}v$ such that `happens(a=v,k)` $\in N$ (and therefor also such that $X \models a[k]{=}v$.
Now, we have that $(\Gamma)^X = (\Gamma_k)^X \cup (\Gamma - \Gamma_k)^X$ by Observation 2 of [SC05b]. The subset $(\Gamma_k)^X$, which is the same as $(\Gamma_k)^{X_0}$, uniquely determines the interpretation of the constants in σ_k, in line with X_0. The subset $(\Gamma - \Gamma_k)^X$, as is obvious from the case analysis above, uniquely determines the interpretation of the constants in $\sigma - \sigma_k$, in line with $X - X_0$. Thus

$$\begin{aligned} X \models_{\mathsf{C}} \Gamma_{k+1}^{D^{Ax,E}} &\cup \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in \mathit{Init}\} \\ &\cup \{a[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N\} \end{aligned}$$

by induction, as desired.
This concludes the inductive step, and so we have our result for all m, by induction. ⌟

Theorem 3.15 Let (Ax, E) be an event calculus system specification. The sequence $(s_0, e_0, s_1, e_1, \ldots, s_m)$ is a run through the transition system defined by $D^{Ax,E}$ iff there exists a stable model M of $(Ax, E, \mathit{Init}, N, \{0, \ldots, m\})$ such that

- $c{=}v \in s_t$ iff `holds_at(c=v,t)` $\in M$, and
- $a{=}v \in e_t$ iff `happens(a=v,t)` $\in N$,

(where $s_0 = \{c{=}v \mid \texttt{initially(c=v)} \in \mathit{Init}\}$ and for all i such that $0 \leqslant i < m$, $a{=}v \in e_i$ iff `happens(a=v,t)` $\in N$).

Proof: This is a straightforward corollary of Theorem 3.14, together with relevant definitions from Section 2.1.4. ⌟

Theorem 3.16 Let (Ax, E) be an event calculus system specification and $P = (Ax, E, \mathit{Init}, N, \{0, \ldots, m\})$ a simplified event calculus program based on (Ax, E). Let M be a Herbrand model of P such that for all t with $0 \leqslant t \leqslant m$, M contains precisely one atom `holds_at(c=v,t)`. Then M is a stable model of $Ax \cup E \cup \mathit{Init} \cup N$ iff there is a stable model M^P of $LP(D^{Ax,E}, m, \mathit{Init}^*, N)$, such that:

- `holds_at(c=v,t)` $\in M$ iff `caused(c=v,t)` $\in M^P$;

- `happens(a=v,t)` $\in M$ iff `happens(a=v,t)` $\in M^P$.

Proof: For the 'only if' direction, first assume that M is as described, and is a stable model of the program $Ax \cup E \cup Init \cup N$. As the narrative N is complete and consistent, then the function $X : \sigma_k \rightarrow \bigcup_{c \in \sigma_k} dom(c)$ defined as

$$\{(c[t], v) \mid (c \in \sigma^f \wedge v \in dom(c) \wedge 0 \leqslant t \leqslant m \wedge \texttt{holds\_at(c=v,t)} \in M) \\ \vee (c \in \sigma^a \wedge v \in dom(c) \wedge 0 \leqslant t < m \wedge \texttt{happens(c=v,t)} \in M)\}$$

is an interpretation of σ_k. So by Theorem 3.14, we have

$$X \models_{\mathsf{C}} \Gamma_m^{D^{Ax,E}} \cup \{c[0]{=}v \Leftarrow \top \mid \texttt{initially(c=v)} \in Init\} \\ \cup \{a[i]{=}v \Leftarrow \top \mid \texttt{happens(a=v,i)} \in N\}.$$

Using Theorem 3.10, we have a stable model M^P of $LP(D^{Ax,E}, m, Init^*, N)$, with the desired properties.
The 'if' part of the proof moves similarly, but in the opposite direction. ⌟

These theorems relating stable models of $\mathcal{EC}+$, runs through the transition system defined by action descriptions, models of causal theories and stable models of simplified event calculus programs are only one part of the work of relating the event calculus to $\mathcal{C}+$ and $\mathcal{EC}+$. In particular, and as we have already stressed, only one of the many variants of the event calculus has been chosen, as a representative of that family of formalisms.

3.11 Summary

In this chapter we have presented an alternative means of working with action descriptions of $\mathcal{C}+$, one that is much more efficient and, we believe, scalable, then that currently in use. Our logic-programmed algorithm takes its inspiration from the Event Calculus, and a working implementation already exists.

We first restricted $\mathcal{C}+$ to $\mathcal{EC}+$. This involved placing constraints on the interaction between defaults and inertia in action descriptions, prohibiting action dynamic laws beyond blanket exogeneity for action constants, and imposing a condition that, apart from defaults, fluent constants should not depend on themselves, in a sense we defined. We gave an account of how signatures, action descriptions, initial states and narratives are represented in our logic programs, and described in detail the rationale behind the axioms we introduce which enable us to reason about narratives. A substantial proof of equivalence was provided, showing that stable models of our programs correspond to runs through the transition system defined according to the semantics of $\mathcal{C}+$. We discussed details of our implementation and the process of checking that the initial state and narrative of actions are consistent, in a sense we explicated, with the action description. Several examples were given, including one standard problem domain discussed in the literature. We gave a systematic comparison of our logic programs to one variant of the event calculus, and also discussed modifications of our implementation designed to avoid the recomputation of answers, based on tabling.

There is another way of conceiving of our work here. We have presented it as a more efficient engine for (a subset of) $\mathcal{C}+$, but the fact that the Event Calculus was taken as the blueprint of how the program should work, suggests that we can see $\mathcal{EC}+$ modern version of one variant of the Event Calculus—and significantly, one which has a very useful semantics of labelled transition systems firmly in place. The two ways of viewing this work gave rise to the overdetermined acronym which is its name: $\mathcal{EC}+$ is the $\mathcal{E}$vent $\mathcal{C}$alculus +, or $\mathcal{E}$fficient $\mathcal{C}$omputation for $\mathcal{C}+$.

Chapter 4

Distant Causation

4.1 Preliminaries

A limitation of $\mathcal{C}+$ as it currently stands is that when writing causal laws one must refer only to states at most one time-step away from each other. It is also impossible directly to say that the performance of an action causes the performance of another action at the next time-step, or at some other time in the future. Yet the fact that an action description of $\mathcal{C}+$ can be viewed as shorthand for an infinite sequence of causal theories (a sequence indexed by the natural numbers), as described in [GLL$^+$04] and rehearsed in Section 2.1.3 of this thesis, together with the fact that there is no limitation on the times to which the rules of those causal theories refer, suggests that one may expand $\mathcal{C}+$, removing the restriction on temporal distance.

There will be benefits. Currently, encoding domains in $\mathcal{C}+$ in which delays and deadlines play an important role is awkward at best and computationally expensive at worst. For example, suppose one wanted to say that 20 time-steps after a *set* action, a *switch* turned to *red*. The signature would at least contain the fluent constant *switch*, with $dom(switch) = \{green, red\}$, and an action constant *set*, with a boolean domain. But how are the causal laws to be written? One might think of something along the following lines:

inertial *switch*,	*timer*=1 **if** $\top$ **after** *set*,
exogenous *set*,	*timer*=2 **if** $\top$ **after** *timer*=1,
default *timer*=*none*,	*timer*=3 **if** $\top$ **after** *timer*=2,
	$\vdots$
	timer=20 **if** $\top$ **after** *timer*=19,
	switch=*red* **if** *timer*=20.

But this do only for one interpretation of the specification. For what if one time-step after the *set* action, another *set* action is performed? Let us suppose that the idea is, not to reset the timer so that countdown begins anew, but rather to ensure that the *switch* is *red* at another time, one step after that triggered by the initial *set* action. It seems obvious that one will need a whole series of $timer_i$ constants, governed by a series of laws which decides when each

of them is to be started, along the following lines:

inertial *switch*,
exogenous a,
$timer_1$=1 **if** $\top$ **after** $set \wedge timer_1$=*none*,
$timer_1$=2 **if** $\top$ **after** $timer_1$=1,
$\vdots$
$timer_1$=20 **if** $\top$ **after** $timer_1$=19,
switch=*red* **if** $timer_1$=20,
default $timer_1$=*none*,
$timer_2$=1 **if** $\top$ **after** $set \wedge \neg timer_1$=*none* $\wedge\ timer_2$=*none*,
$timer_2$=2 **if** $\top$ **after** $timer_2$=2,
$\vdots$
switch=*red* **if** $timer_2$=20,
default $timer_2$=*none*,
$timer_3$=1 **if** $\top$ **after** $set \wedge \neg timer_1$=*none* $\wedge\ \neg timer_2$=*none*
$\wedge\ timer_3$=*none*,
$timer_{21}$=1 **if** $\top$ **after** $set \wedge \neg timer_1$=*none* $\wedge \cdots \wedge \neg timer_{20}$=*none*
$\wedge\ timer_{21}$=*none*,
default $timer_{21}$=*none*

That will now give the results we wanted. But it is hardly a perspicuous representation of the behaviour of the system, and though one might invent macros to generate such causal laws, there will clearly be a computational penalty to pay in the implicit causal theories with which CCalc, for example, operates.

This chapter presents a solution. (An earlier version of the work presented in this chapter was published as [CS05].)

4.2 Times

One could change the syntax of $\mathcal{C}+$ as it stands in the following way. Let λ be conceived of as an operator which enables one to refer to the immediately preceding state. Fluent dynamic laws can then be written as

$$F \textbf{ if } G \wedge \lambda(H);$$

both action dynamic laws and static laws will be written in the form

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

which is the same as for $\mathcal{C}+$.

We will allow ourselves to nest the λ. In this extended language, signatures are defined as before for $\mathcal{C}+$, and causal laws have the form

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

where F is as before a formula of the signature, and G is given by

$$G \ ::= \ c{=}v \mid \top \mid \neg G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \lambda(G). \tag{4.1}$$

We can write $\lambda\lambda(F)$ as $\lambda^2(F)$, and so on. We insist that if the left-hand side of a causal law contains a fluent constant, then no action constant should appear on the right-hand side outside the scope of any λ. Also, we stipulate that if there is a λ with index greater than 0 on the right-hand side of a law, then that law's left-hand side must not contain statically defined fluents. These restrictions are essentially inherited from those in $\mathcal{C}+$ pertaining to action dynamic laws and statically determined fluents.

It is easy to show that causal rules defined as above have the following *canonical form*:

$$F \textbf{ if } \lambda^{n_0}(G_0) \wedge \cdots \wedge \lambda^{n_k}(G_k), \tag{4.2}$$

where $G_0, \ldots, G_n$ are formulas of $\sigma^f \cup \sigma^a$ (i.e., formulas containing no λ), and

- $k \geqslant 0$;
- $(n_0, \ldots, n_k)$ is a strictly increasing sequence of non-negative integers;
- if F contains fluent constants and $n_0 = 0$, then G_0 contains no action constants;
- if F contains statically determined fluent constants, then there is no λ-index greater than 0 on the right-hand side.

We have been calling the $n_0, \ldots, n_k$ λ-*indices*; we say that a law (4.2) has a greatest λ-index of n_k. We also allow ourselves to drop any operator λ^0, and to remove any λ-index of 1 from a causal law. An *action description* is a set of causal laws. This extension of $\mathcal{C}+$ is called $\mathcal{C}+_{timed}$.

To every action description D (signature $\sigma^f \cup \sigma^a$) and non-negative integer t there corresponds a causal theory Γ_t^D. The signatures of these causal theories are defined in the same way as for $\mathcal{C}+$. The causal laws of Γ_t^D are

$$F[i + n_k] \Leftarrow G_0[i + (n_k - n_0)] \wedge \cdots \wedge G_k[i + (n_k - n_k)],$$

or more briefly,

$$F[i + n_k] \Leftarrow \bigwedge_{j=0}^{k} G_j[i + (n_k - n_j)],$$

for every causal law in D and every

- $i \in \{0, \ldots, t - n_k - 1\}$, if F contains an action constant,
- $i \in \{0, \ldots, t - n_k\}$, otherwise;

we also include

$$c[0]{=}v \Leftarrow c[0]{=}v,$$

for every simple fluent constant c and $v \in dom(c)$. So much for the language definition and translation into the language of causal theories. We allow ourselves to use the syntax and abbreviations of $\mathcal{C}+$ laws (those in Section 2.1.6) in obvious ways: so that

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

ought to be understood as meaning

$$F \textbf{ if } G \wedge \lambda(H),$$

and so forth.

For the purpose of illustration, consider the following simple domain DS, with Boolean signature $\sigma^f = \{p\}$ (p is simple) and $\sigma^a = \{a\}$. The causal laws are:

$$\begin{aligned} &\textbf{inertial } p, \\ &\textbf{exogenous } a \\ &p \textbf{ if } \lambda^2(a), \end{aligned}$$

where the following abbreviation holds, as in $\mathcal{C}+$:

$$\begin{aligned} \textbf{inertial } c &\mapsto c{=}v \textbf{ if } c{=}v \wedge \lambda(c{=}v), && \text{for all } v \in dom(c),\ (c \in \sigma^f) \\ \textbf{exogenous } a &\mapsto a{=}v \textbf{ if } a{=}v, && \text{for all } v \in dom(c),\ (a \in \sigma^a) \end{aligned}$$

For time-index 2, the causal theory determined by the action description DS is:

$$\begin{aligned} a[0]{=}\mathbf{t} &\Leftarrow a[0]{=}\mathbf{t}, & p[1]{=}\mathbf{t} &\Leftarrow p[1]{=}\mathbf{t} \wedge p[0]{=}\mathbf{t}, \\ a[0]{=}\mathbf{f} &\Leftarrow a[0]{=}\mathbf{f}, & p[1]{=}\mathbf{f} &\Leftarrow p[1]{=}\mathbf{f} \wedge p[0]{=}\mathbf{f}, \\ a[1]{=}\mathbf{t} &\Leftarrow a[1]{=}\mathbf{t}, & p[2]{=}\mathbf{t} &\Leftarrow p[2]{=}\mathbf{t} \wedge p[1]{=}\mathbf{t}, \\ a[1]{=}\mathbf{f} &\Leftarrow a[1]{=}\mathbf{f}, & p[2]{=}\mathbf{f} &\Leftarrow p[2]{=}\mathbf{f} \wedge p[1]{=}\mathbf{f}. \\ p[2]{=}\mathbf{t} &\Leftarrow a[0]{=}\mathbf{t}, \\ p[0]{=}\mathbf{t} &\Leftarrow p[0]{=}\mathbf{t}, \\ p[0]{=}\mathbf{f} &\Leftarrow p[0]{=}\mathbf{f}, \end{aligned}$$

One may then find the models of this theory using the 'literal completion' method, as described in Section 2.1.5; CCalc can do this for us. A sample model of Γ_2^{DS} is shown in Figure 4.1. It is straightforward to adapt CCalc to

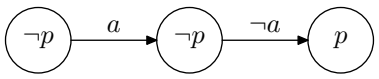

Figure 4.1: A model of Γ_2^{DS}

accept inputs of action descriptions written in the extended language of $\mathcal{C}+_{timed}$, and to effect the translation into causal theories as described above.

4.3 Graphical Models

4.3.1 Run Systems

It remains to decide what to do about the transition systems which were the original semantics for $\mathcal{C}+$. For whilst one could understand $\mathcal{C}+_{timed}$ simply as a

means of writing in abbreviated form a family of causal theories, a very attractive feature of $\mathcal{C}+$ is that action descriptions can be shown to define labelled transition systems. These transition systems may afford a useful connection between $\mathcal{C}+$ and other formalisms for reasoning about actions, and of course they are useful in their own right as aids to visualization. If we look at the transition systems defined by $\mathcal{C}+_{timed}$ theories, however, we see that important information is lacking.

Recall the means of calculating the transition system for an action description of $\mathcal{C}+$ described in Section 2.1.4. If we apply the definitions given there (specifically, Definition 2.6, which defines states and transitions in terms of models of Γ_0^D and Γ_1^D) to action descriptions of $\mathcal{C}+_{timed}$, and calculate the transition system for the domain DS, the result is as shown in Figure 4.2. However, that

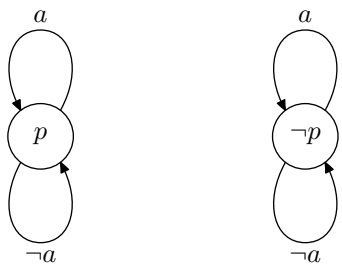

Figure 4.2: The (flawed) $\mathcal{C}+$-style transition system for $\mathcal{C}+_{timed}$ domain DS

is clearly flawed, for the run of the system depicted in Figure 4.1 cannot be traced in the diagram. The reason for the failure is clear: the causal theory Γ_1^D used to find the transitions has not taken into account the law p **if** $\lambda^2(a)$ of our action description. In general, the theorem correlating paths through the transition system with models of causal theories (Theorem 2.10) does not hold for $\mathcal{C}+_{timed}$.

Accordingly, we broaden our conception and define a kind of transition system called a *run system*. These are different from the subclass of transition systems defined by $\mathcal{C}+$ action descriptions, in two respects. First, states (understood as models of Γ_0^D) may be represented by more than one vertex. Secondly, out of each set of vertices which represent the same state, a privileged *initial* vertex will be marked.

Definition 4.1 A *(labelled) run system* of a signature $\sigma^f \cup \sigma^a$ is any graph G with vertices $V(G)$ and edges $E(G)$, such that:

- $V(G) \subseteq \mathrm{I}(\sigma^f) \times \mathbb{N}$,
- $E(G) \subseteq V(G) \times \mathrm{I}(\sigma^a) \times V(G)$.

Any $(s, 0) \in V(G)$ is called an *initial* vertex. ⌟

In contrast with the transition systems defined by $\mathcal{C}+$ action descriptions, vertices are pairs of states and natural numbers (the use of $\mathbb{N}$ here is arbitrary, and is simply a means of allowing more than one vertex to be associated with a given state). Where (s, n) is a vertex of a run system, we call s the *state component*, and if $((s, n), e, (s', n'))$ is an edge of a run system, then we call (s, e, s') the *transition component*.

Definition 4.2 A labelled run system G is said to *represent* an action description D of $\mathcal{C}+_{timed}$ (with signature σ) when, for all integral t with $t \geqslant 0$

$$s_0[0] \cup e_0[0] \cup s_1[1] \cup \cdots \cup e_{t-1}[t-1] \cup s_t[t]$$

(where the $s_i \in \mathrm{I}(\sigma^f)$ and the $e_i \in \mathrm{I}(\sigma^a)$) is a model of Γ_t^D iff there is a path

$$((s_0, 0), e_0, (s_1, n_1), \ldots, e_{t-1}, (s_t, n_t))$$

through G. ⌟

For any action description of $\mathcal{C}+_{timed}$ there clearly exists at least one run system representing it. (Consider trees whose roots are the initial vertices, and where paths of length t correspond one-to-one with models of Γ_t^D.)

In diagrams, we will represent the initial states by circling them twice, and we will not include the n component of our vertices (s, n). As an illustration of what we are working towards, a run system for the domain DS is shown in Figure 4.3. It can clearly be seen that paths through this run system beginning

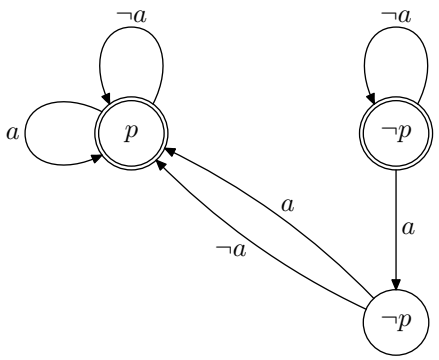

Figure 4.3: Run system for the simple action description DS

at one of the twice-circled vertices correspond to models of the causal theories generated by DS.

We now introduce the notion of a 'commitment', which will be of central importance in the generation of run systems from action descriptions of $\mathcal{C}+_{timed}$.

4.3.2 Commitments

The idea is that as a system makes runs, it may accrue commitments, which express that something should be true in the future, if certain other things are true. (Thus we use 'commitment' here as a technical term, unrelated to the concept encountered in, for example, multi-agent systems.) Commitments have the same syntax as causal rules, to which they have a very close relationship.

Figure 4.4 represents a model of the causal theory Γ_2^{DS} (the same model shown in Figure 4.1). That model, as we know, can be partitioned into

$$s_0[0] \cup e_0[0] \cup \cdots \cup s_2[2]$$

and in our diagram the vertices are the s_i, and the edges (s_i, e_i, s_{i+1}). To the right of each vertex, a set of commitments has been drawn. These commitments

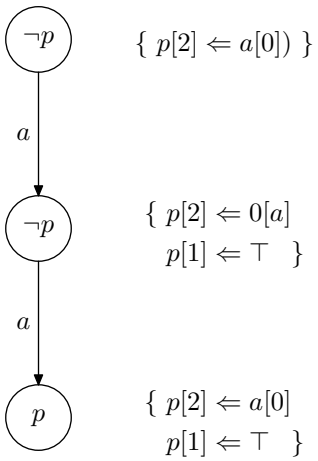

Figure 4.4: A model of Γ_2^{DS}, marked with commitments

stem from the law p **if** $\lambda^2(a)$ in the action description DS. The first thing to notice is that vertices which encode the same interpretation of σ^f may be labelled by different sets of commitments: this is true of the first and second vertices in the diagram. From this flows the fact that in the run system depicted in Figure 4.3, there are two vertices which are labelled with $\neg p$.

In commitments, the time-stamp is to be understood as relative to the current state—i.e., relative to the one which is labelled by the commitment. Thus the fact that the second vertex in our diagram is labelled by the commitment $p[2] \Leftarrow a[0]$ means that if a is performed at the outgoing edge, then 2 time-steps into the future, p must be true.

Starting at the top and moving down through the diagram, we see that if a is performed on the outgoing edge of the first state, p must be true two time-steps later. At the next state this is also true; further, since a *was* performed between states one and two, p is now definitely constrained to be true at the third state. The third state itself is labelled by the same commitments as the second, and so on.

In general, there are two ways in which in which a state in such a diagram may come to be labelled by a commitment: because of the nature of the state itself, and because a commitment has been inherited from a previous state.

Consider a model of some causal theory Γ_t^D. If a state s_j of that model is labelled with the commitment

$$F[n] \Leftarrow G_0[n_0] \wedge \cdots \wedge G_k[n_k]$$

(where here $n \geqslant n_0$), then this should be taken to mean that if

$$\begin{gathered} s_{j+n_k} \cup e_{j+n_k} \models G_k, \\ \vdots \\ s_{j+n_0} \cup e_{j+n_0} \models G_0, \end{gathered}$$

then we must have $s_{j+n} \cup e_{j+n} \models F$.

We define a function from commitments to sets of commitments, which will be used in specifying how these commitments change from one state to another. Let x be the commitment

$$F[n] \Leftarrow G_0[n_0] \wedge \cdots \wedge G_k[n_k].$$

The value of $cmt(x)$ will be

$$\{\ F[n-1] \Leftarrow G_0[n_0-1] \wedge \cdots \wedge G_k[n_k-1]\ \}$$

if $n-1>0$, or if $n-1=0$ and F is an action atom; we omit any conjuncts of the right-hand side which have time-stamps less than 0. Otherwise, the value of $cmt(x)$ is the empty set. Thus the function cmt has as its domain the set of commitments of some signature, and has as its range the set containing singletons of commitments, together with the empty set. As an example, we refer back to the domain DS:

$$cmt(p[2] \Leftarrow a[0]) = \{p[1] \Leftarrow \top\},$$
$$cmt(p[1] \Leftarrow \top) = \emptyset,$$

By convention, if S is a set of commitments, we set

$$cmt(S) = \bigcup_{x \in S} cmt(x).$$

4.3.3 Generation of Run Systems

Let D be an action description of $\mathcal{C}+_{timed}$, with signature $\sigma^f \cup \sigma^a$. Assume the laws of D are in canonical form, and so have the structure

$$F \textbf{ if } \lambda^{n_0}(G_0) \wedge \cdots \wedge \lambda^{n_k}(G_k) \tag{4.3}$$

for some sequence of natural numbers $(n_0, \ldots, n_k)$. Let $\overline{D} \subseteq D$ be the set of all those laws of D which have a maximum λ-index greater than 1, or else have this index equal to 1 and an action atom as F. Thus $\overline{D}$ contains those laws of D which could not be expressed in $\mathcal{C}+$, and $D - \overline{D}$ is, despite its odd syntax, a $\mathcal{C}+$ action description (with the same signature).

Let *init* be the set

$$\{s \in \mathrm{I}(\sigma^f) \mid s[0] \models_{\mathsf{C}} \Gamma_0^D\}.$$

init can be seen as being the set of states (a notion inherited from $\mathcal{C}+$) of our system, the possible left-hand parts of the pairs which are the vertices of our run systems. Let the set com_D contain those commitments

$$F[n_k] \Leftarrow G_0[n_k - n_0] \wedge \cdots \wedge G_k[n_k - n_k]$$

such that there is a law of form (4.3) in $\overline{D}$ (the last conjunct here is of course simply $G_k[0]$).

Run systems, it will be recalled, have as their vertices pairs consisting of a state (a member of *init*)and a natural number. Before systems of this kind can be generated, we use sets of commitments as the second components. So, we

make a graph G_D, which has as its set of vertices those pairs (s, c), where s is as usual an interpretation of σ^f, and where c is such that

$$com_D \subseteq c \subseteq \bigcup_{n \geqslant 0} cmt^n(com_D).$$

The set $\bigcup_{n \geqslant 0} cmt^n(com_D)$ includes all commitments which could label any state.

For the edges, we need to introduce a function which will model how commitments change over time as a consequence of the preceding state of, and actions performed in, a transition. We know that all commitments which can label a state are contained in

$$\bigcup_{n \geqslant 0} cmt^n(com_D);$$

commitments in general have the form

$$F[n] \Leftarrow G_0[n_0] \wedge \cdots \wedge G_k[n_k].$$

The function introduced is $trans_{com}$, which has the domain $\mathrm{I}(\sigma^f) \times \mathrm{I}(\sigma^a) \times \bigcup_{n \geqslant 0} cmt^n(com_D)$, where, if z is a commitment:

$$trans_{com}(s, e, z) = \begin{cases} \emptyset & \text{if } n_j = 0 \text{ and } s \cup e \not\models G_j \\ cmt(z) & \text{otherwise.} \end{cases}$$

Clearly, we want commitments to persist (time-stamps decremented) through a transition, only when those formulas on the right-hand side of the commitment which relate to the transition are satisfied. In our example we have that

$$\begin{aligned} trans_{com}(\{\neg p\}, \{a\}, p[2] \Leftarrow a[0]) &= cmt(p[2] \Leftarrow a[0]), \\ &= \{p[1] \Leftarrow \top\}. \end{aligned}$$

Thus we ensure the presence of the right commitments. By convention, where c is a set of commitments, we let

$$trans_{com}(s, e, c) =_{def} \bigcup_{z \in c} trans_{com}(s, e, z).$$

In order to ensure that the transition components (s, e, s') of our graph respect the commitments which relate to them, we have include those commitments in the conditions which define the edges. So, we insist that the vertices of our graphs G of the form $((s, c), e, (s', c'))$ must have

$$s[0] \cup e[0] \cup s'[1] \models_{\mathsf{C}} \Gamma_1^D \cup c^*,$$

where c^* includes those members of c all of whose constants are members of σ_1.

Definition 4.3 Let D be an action description of $\mathcal{C}+_{timed}$, with signature σ. The *preliminary run system* G_D defined by D is the graph whose vertices $V(G_D)$ are the pairs (s, c) such that

- $s \in \mathrm{I}(\sigma^f)$ and $s[0] \models_{\mathsf{C}} \Gamma_0^D$; and
- $com_D \subseteq c \subseteq \bigcup_{n \geqslant 0} cmt^n(com_D)$.

The edges $E(G_D)$ of the graph are those triples $((s,c),e,(s',c'))$ such that

- $(s,c),(s',c') \in V(G_D)$;
- $c' = com_D \cup trans_{com}(s,e,c)$;
- $s[0] \cup e[0] \cup s'[1] \models_{\mathsf{C}} \Gamma_1^D \cup c^*$, where c^* is the result of removing from c any commitments which employ a constant not in the signature σ_1. ⌟

To obtain a run system for D from a preliminary run system, all that remains is to rename the second components of the vertices: to do this, we simply have to make a suitable correspondence between sets of commitments and non-negative integers, such that the set com_D corresponds to 0. This can easily be done. Once the renaming of vertices and edges has been effected, we obtain G_D^*, which is a run system representing D in the sense introduced in Definitions 4.1 and 4.2.

A transition system T, of the type defined by a $\mathcal{C}+$ action description, is essentially a run system in which there are only initial states, so that $V(T) = \{(s,0) \in V(T)\}$.

It remains, of course, to prove that this method of generating run systems works. The desired result states that given an action description D of $\mathcal{C}+_{timed}$, paths through the system G_D^* of length t which begin at an initial vertex $(s,0)$ correspond to models of the causal theory Γ_t^D. In other words, we need to show that G_D^* represents the domain D.

Theorem 4.4 Let D be an action description of $\mathcal{C}+_{timed}$ with signature $\sigma^f \cup \sigma^a$, and t a non-negative integer. Then, where $s_0,\ldots,s_t$ are interpretations of σ^f, and $e_0,\ldots,e_{t-1}$ interpretations of σ^a, we have that

$$s_0[0] \cup e_0[0] \cup \cdots \cup s_t[t]$$

is a model of Γ_t^D iff

$$((s_0,0),e_0,(s_1,n_1),\ldots,e_{t-1},(s_t,n_t))$$

is a path through G_D^*, for some $n_1,\ldots,n_t \in \mathbb{N}$.

Proof: Given the way that G_D^* is derived from G_D, we can show our theorem holds by proving that

$$s_0[0] \cup e_0[0] \cup \cdots \cup s_t[t] \models_{\mathsf{C}} \Gamma_t^D$$

iff

$$((s_0,com_D),e_0,(s_1,c_1),\ldots,e_{t-1},(s_t,c_t))$$

is a path through G_D. This we do by induction on the length t of run.

(Base case: $t = 0$.) Assume that $s_0[0] \models_{\mathsf{C}} \Gamma_t^D$; then clearly $(s_0, com_D) \in V(G_D)$ by the definition of the vertices of G_D. Alternately, if $((s_0, com_D))$ is a path through G_D, then we must have $s_0[0] \models_{\mathsf{C}} \Gamma_0^D$.

(Inductive step: assume true for $t = k$, show for $t = k+1$.) We assume the result for $t = k$. Then, for the 'only if' direction of the bi-implication, assume further that

$$s_0[0] \cup e_0[0] \cup \cdots \cup s_k[k] \cup e_k[k] \cup s_{k+1}[k+1] \models_{\mathsf{C}} \Gamma_{k+1}^D.$$

We will abbreviate this model of Γ^D_{k+1} by I^{k+1}, and we will abbreviate the restriction of I^{k+1} to σ_k by I^k. Now clearly the causal theory Γ^D_{k+1} can be thought of as divided into two parts: Γ^D_k, all of whose constants are contained in the signature σ_k, and $(\Gamma^D_{k+1} - \Gamma^D_k)$, whose constants are in σ_{k+1}, but the heads of whose rules contain no constant not in $(\sigma_{k+1} - \sigma_k)$. By assumption, $I^{k+1} \models_{\mathsf{C}} \Gamma^D_{k+1}$, which means that I^{k+1} is the unique model of $(\Gamma^D_{k+1})^{I^{k+1}}$. Now by Observation 2 of [SC05a],

$$\begin{aligned}(\Gamma^D_{k+1})^{I^{k+1}} &= (\Gamma^D_{k+1} - \Gamma^D_k)^{I^{k+1}} \cup (\Gamma^D_k)^{I^{k+1}} \\ &= (\Gamma^D_{k+1} - \Gamma^D_k)^{I^{k+1}} \cup (\Gamma^D_k)^{I^k}.\end{aligned}$$

(The second line follows from the considerations above about the constants in rules of Γ^D_k.) The set $(\Gamma^D_{k+1} - \Gamma^D_k)^{I^{k+1}}$ contains only constants from $(\sigma_{k+1} - \sigma_k)$ and therefore this alone must determine $I^{k+1} - I^k$ as its unique interpretation; similarly, $(\Gamma^D_k)^{I^k}$ contains only constants in σ_k, and this must determine I^k as its unique interpretation. From the latter we immediately conclude that

$$s_0[0] \cup e_0[0] \cup \cdots \cup s_k[k] \models_{\mathsf{C}} \Gamma^D_k,$$

and thus by the inductive hypothesis we have that

$$((s_0, com_D), e_0, (s_1, c_1), \ldots, e_{k-1}, (s_k, c_k))$$

is a run through G_D. It remains to show that this run can be extended to one of the right form, by adding $(e_k, (s_{k+1}, c_{k+1}))$ (for an appropriate c_{k+1}) to the end. It is clear which set of commitments we need, and so we define

$$c_{k+1} = com_D \cup trans_{com}(s_k, e_k, c_k).$$

We now must show that $((s_k, c_k), e_k, (s_{k+1}, c_{k+1}))$ is an edge of G_D. First, for it to be true that $(s_{k+1}, c_{k+1}) \in V(G_D)$, we need that $s_{k+1}[0] \models_{\mathsf{C}} \Gamma^D_0$ and that

$$com_D \subseteq c_{k+1} \subseteq \bigcup_{n \geqslant 0} cmt^n(com_D).$$

The latter is immediate from the definition of c_{k+1}. A straightforward extension of Proposition 7 from [GLL+04] shows that s_{k+1} is a 'state' in the sense of $\mathcal{C}+$, i.e. that $s_{k+1}[0] \models_{\mathsf{C}} \Gamma^D_0$. Thus $(s_{k+1}, c_{k+1}) \in V(G_D)$. In order to prove that the triple $((s_k, c_k), e_k, (s_{k+1}, c_{k+1}))$ is an edge, we have to show that

$$s_k[0] \cup e_k[0] \cup s_{k+1}[1] \models_{\mathsf{C}} \Gamma^D_1 \cup c^*_k,$$

where c^*_k is the result of removing from c_k any commitments which employ a constant not in σ_1. Let I denote

$$s_k[0] \cup e_k[0] \cup s_{k+1}[1],$$

We have to show that the unique interpretation satisfying $(\Gamma^D_1 \cup c^*_k)^I$ is I. Clearly the interpretation of the simple fluent constants of the form $c[i]$ is uniquely determined in $(\Gamma^D_1 \cup c^*_k)^I$: the presence of the rules

$$c[0]{=}v \Leftarrow c[0]{=}v$$

which flows from the exogeneity of the initial state of runs implies that. And there can be no rule

$$F[0] \Leftarrow X$$

in $\Gamma_1^D \cup c_k^*$ whose body is true in I but whose head is false, where F contains fluent constants: if such a rule stemmed from a static law of D, then we would have a rule $F[k] \Leftarrow X[+k]$ in Γ_{k+1}^D, with $I^{k+1} \models X[+k] \wedge \neg F[k]$, which is impossible. On the other hand, if the rule $F[0] \Leftarrow X$ were in c_k^*, then given how the set c_k is constructed, there would be a rule $F[k] \Leftarrow X'$ in Γ_{k+1}^D, whose body was true and whose head was false in I^{k+1}, which again, is impossible. So $(\Gamma_1^D \cup c_k^*)^I$ uniquely determines the interpretation of the simple fluent constants $c[0]$, in accordance with I itself. Similar argumentation shows that the interpretation of the other members of σ_1 is uniquely determined, in line with $s_k[0] \cup e_k[0] \cup s_{k+1}[1]$, and thus that this interpretation is a model of $\Gamma_1^D \cup c_k^*$. In fact, it is not difficult to see that $((\Gamma_1^D \cup c_k^*)^I)[+k]$ and that portion of $(\Gamma_{k+1}^D)^{I^{k+1}}$ which has constants $c[k]$ or $c[k+1]$ must contain the same formulas, *apart* from those which stem in $(\Gamma_{k+1}^D)^{I^{k+1}}$ from the presence of regular fluent dynamic laws and which govern the transition (s_{k-1}, e_{k-1}, s_k). But the formulas which stem from the rules derived from these laws can only contain simple fluent constants $c[k]$, and *these* constants, as has been argued above, are uniquely determined in $(\Gamma_1^D \cup c_k^*)^I$ thanks to the rules

$$c[0]{=}v \Leftarrow c[0]{=}v.$$

So, we must have that the triple $((s_k, c_k), e_k, (s_{k+1}, c_{k+1}))$ is an edge of the graph G_D, which is what we needed to show. So we have that

$$((s_0, com_D), e_0, \ldots, (s_k, c_k), e_k, (s_{k+1}, c_{k+1}))$$

is a path through G_D.
For the 'if' part, assume that

$$((s_0, com_D), e_0, (s_1, c_1), \ldots, e_k, (s_{k+1}, c_{k+1}))$$

is a run through G_D. Then so is

$$((s_0, com_D), e_0, (s_1, c_1), \ldots, e_{k-1}, (s_k, c_k)),$$

and so by the induction hypothesis we have

$$s_0[0] \cup e_0[0] \cup \cdots \cup s_k[k] \models_{\mathsf{C}} \Gamma_k^D.$$

As $((s_k, c_k), e_k, (s_{k+1}, c_{k+1}))$ is an edge of the graph G_D, we also have, by definition, that

- $c_{k+1} = com_D \cup trans_{com}(s_k, e_k, c_k)$ and
- $s_k[0] \cup e_k[0] \cup s_{k+1}[1] \models_{\mathsf{C}} \Gamma_1^D \cup c_k^*$.

By an argument which is essentially the reverse of that used for the 'only if' part of this proof, we can show that I^{k+1} (using terminology we introduced in the first part of the proof) is a model of Γ_{k+1}^D. That concludes the 'if' part.
And so, on the assumption that the bi-implication holds for $t = k$, we have shown it for $t = k + 1$. By induction we conclude that it holds for all $t \geqslant 0$. ⌟

4.3.4 An Example Generation

Let us work through the simple action description DS of $\mathcal{C}+_{timed}$, in order to illustrate the procedures described above. The domain is Boolean, has signature $\sigma^f \cup \sigma^a$, and contains the laws

$$\begin{aligned}&\textbf{inertial } p,\\ &\textbf{exogenous } a,\\ &p \textbf{ if } \lambda^2(a).\end{aligned}$$

Thus we have $\overline{DS} = \{p \textbf{ if } \lambda^2(a)\}$.

The causal theory Γ_0^{DS} is evidently

$$\begin{aligned}p &\Leftarrow p,\\ \neg p &\Leftarrow \neg p,\end{aligned}$$

so that the set $init$ contains the two states

$$\{p\} \quad \text{and} \quad \{\neg p\}.$$

We also have com_{DS} as the singleton containing the commitment $p[2] \Leftarrow a[0]$. As was noted previously, we have

$$\begin{aligned}cmt(p[2] \Leftarrow a[0]) &= \{p[1] \Leftarrow \top\},\\ cmt(p[1] \Leftarrow \top) &= \emptyset,\end{aligned}$$

so that

$$\bigcup_{n \geqslant 0} cmt^n(com_{DS}) = \{p[2] \Leftarrow a[0], p[1] \Leftarrow \top\}.$$

We now make the graph G_{DS}. The vertices of G_{DS} are the pairings of members of $init$ with those subsets of $\bigcup_{n \geqslant 0} cmt^n(com_{DS})$ which also contain com_{DS}, namely:

$$\begin{aligned}&(\{p\},\{p[2] \Leftarrow a[0]\}),\\ &(\{p\},\{p[2] \Leftarrow a[0], p[1] \Leftarrow \top\}),\\ &(\{\neg p\},\{p[2] \Leftarrow a[0]\}),\\ &(\{\neg p\},\{p[2] \Leftarrow a[0], p[1] \Leftarrow \top\}),\end{aligned}$$

To find the edges for our graph, we first need the causal theory Γ_1^{DS}. This has the laws:

$$\begin{aligned}p[1] &\Leftarrow p[1] \wedge p[0],\\ \neg p[1] &\Leftarrow \neg p[1] \wedge \neg p[0],\\ a[0] &\Leftarrow a[0],\\ \neg a[0] &\Leftarrow \neg a[0],\\ p[0] &\Leftarrow p[0],\\ \neg p[0] &\Leftarrow \neg p[0].\end{aligned}$$

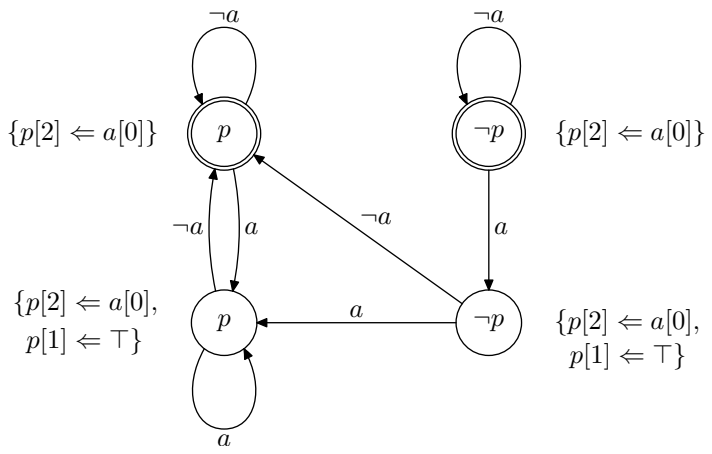

Figure 4.5: The graph G_{DS}, with states and associated commitments

The triples $((s, c), e, (s', c'))$ which satisfy the constraints (given in the previous section) on edges of G_{DS} will not be calculated explicitly; they are represented in Figure 4.5, which shows G_{DS}. In the diagram, the vertices (s, c) have been depicted so that the components s are shown inside circles, with the commitments c shown adjacently and outside. Vertices of the form (s, com_{DS}) have been circled twice.

After replacing the commitments by natural numbers, we arrive at the run system G^*_D, whose vertices are

$$(\{p\}, 0), (\{p\}, 1), (\{\neg p\}, 0), (\{\neg p\}, 1),$$

and whose edges are

$$\begin{array}{ll}
((\{p\}, 0), \{a\}, (\{p\}, 1)), & ((\{\neg p\}, 0), \{a\}, (\{\neg p\}, 1)), \\
((\{p\}, 0), \{\neg a\}, (\{p\}, 0)), & ((\{\neg p\}, 0), \{\neg a\}, (\{\neg p\}, 0)), \\
((\{p\}, 1), \{a\}, (\{p\}, 1)), & ((\{\neg p\}, 1), \{a\}, (\{p\}, 1)), \\
((\{p\}, 1), \{\neg a\}, (\{p\}, 0)), & ((\{\neg p\}, 1), \{\neg a\}, (\{p\}, 0)),
\end{array}$$

It is clear that paths through this run system—beginning at a double-circled vertex and of length t—correspond to models of the causal theory Γ^{DS}_t, as Theorem 4.4 assures us.

4.3.5 Second Example Generation

As a further example, consider the following, more complex action description of $\mathcal{C}+_{timed}$, which we call D^{pq}. The Boolean signature contains only the simple fluent constants p and q.

inertial p
caused p **if** $\lambda^2(q)$
caused $\neg q$ **if** $\lambda(q)$
caused $\neg q$ **if** $\lambda(\neg q)$

It is to be hoped that the behaviour which this action description defines is intuitively obvious: if q is true in a state, then two states into the future, p must be true. Otherwise, inertia determines the behaviour of p. As for q, it may be either true or false in the initial state, but in all states after the initial, it must be false. There are no action constants.

The laws of this action description have been chosen to illustrate how the graphs G_D and G_D^* (which, of course, are isomorphic) may often be pruned without removing the property that they are representative of the relevant action description. (Perhaps it was obvious, in the case of the action description in Section 4.3.4, that the run systems can be reduced: for the system which resulted there, and which was depicted in Figure 4.5, has more vertices than the run system in Figure 4.3 which represents the same system.)

For the action description D^{pq}, we have that the set *init*, which is given by $\{s \in \mathrm{I}(\sigma^f) \mid s[0] \models_{\mathsf{C}} \Gamma_0^{D^{pq}}\}$, is equal to

$$\{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}\},$$

and thus all possible combinations of evaluations of p and q are present. The set $com_{D^{pq}}$ evidently contains only the single commitment

$$p[2] \Leftarrow q[0],$$

and so we have

$$\bigcup_{n \geqslant 0} cmt^n(com_{D^{pq}}) = \{p[2] \Leftarrow q[0],\ p[1] \Leftarrow \top\}$$

The vertices of the graph $G_{D^{pq}}$ are then pairs where the first place is taken by a member of *init* and the second by a subset of $\bigcup_{n \geqslant 0} cmt^n(com_{D^{pq}})$ which contains $p[2] \Leftarrow q[0]$. There are clearly 8 such combinations, and thus $|V(G_{D^{pq}})| = 8$.

As for the edges of the graph $G_{D^{pq}}$, they are triples $((s, c), e, (s', c')$ such that

- $s[0] \cup e[0] \cup s'[1] \models_{\mathsf{C}} \Gamma_1^{D^{pq}} \cup c^*$, where $c*$ is the result of removing from c all commitments in which there is a constant not in the signature σ_1; and
- $c' = com_{D^{pq}} \cup trans_{com}(s, e, c)$.

After an analysis to obtain the set of edges $E(G_{D^{pq}})$, we can depict the full graph $G_{D^{pq}}$ as in Figure 4.6. We have used a concise representation for the vertices of the graph, abbreviating the two commitments $p[2] \Leftarrow q[0]$ and $p[1] \Leftarrow \top$ by the numerals **2** and **1**. Brackets for sets and tuples have also been omitted. As normal, those states of the form $(s, com_{D^{pq}})$ have been circled twice, denoting the fact that they are the legitimate starting point of runs through the system.

We have also shaded those nodes which can be reached on a path starting from an 'initial node' (one which is of the form (s, com_D)). In our previous generation of a run system, it all vertices in the graph could be so reached, but, as mentioned above, in the current instance we have chosen causal laws specifically so that some nodes may never be reached. Clearly, these nodes can be removed from G_D^* (or G_D), along with any edges incident to them (we have drawn these edges with dashed lines in Figure 4.6). It is easy to see why, given the causal laws of the action description, the top outer nodes cannot be reached:

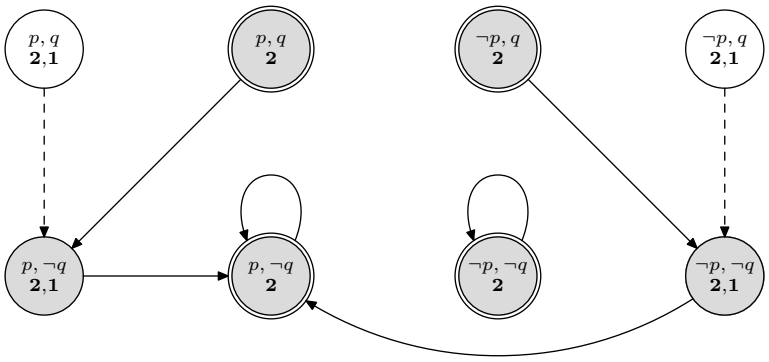

Figure 4.6: The graph $G_{D^{pq}}$

for the presence in each of the commitment $p[1] \Leftarrow \top$ implies that in a previous state, q would be true: but there can be no edge leading to these nodes (and so no such previous state) as any state component of a node reachable from another must have q as false, given the law $\neg q$ **if** $\lambda(q)$.

4.3.6 Reduction

We know that if we remove the unshaded nodes and dashed edges from Figure 4.6 then the system depicted continues to represent D^{pq}; this is true in general.

Theorem 4.5 Let D be an action description of $\mathcal{C}+_{timed}$. Then if $X \subseteq V(G_D^*)$ is the subset of nodes (s_t, c_t) of G_D such that for no $t \geqslant 0$ is there a path

$$((s_0, 0), e_0, \ldots, e_{t-1}, (s_t, n_t)),$$

then the graph G_D^- whose vertices are $V(G_D^*) - X$ and whose edges are

$$E(G_D^*) - \{((s, n), e, (s', n')) \mid (s, n) \in X \text{ or } (s', n') \in X\}$$

is a run system representing D.

Proof: Let G_D^* represent D, and first assume that

$$s_0[0] \cup \cdots \cup s_t[t] \models_{\mathsf{C}} \Gamma_t^D.$$

Then there is a path through G_D^* of the form

$$((s_0, 0), e_0, \ldots, (s_t, n_t)),$$

and clearly this path must also be in G_D^-. The other direction is just as trivial.⌟

The removal of these nodes and edges suggests that we impose an ordering on run systems representing a given system described by $\mathcal{C}+_{timed}$, and it seems that the most appropriate ordering to consider is one based on the cardinality of vertices and edges.

Definition 4.6 Let D be an action description of $\mathcal{C}+_{timed}$, and G^1_D, G^2_D be run systems representing D. Then we define an ordering such that:

$$G^1_D <_r G^2_D \quad \text{if} \quad \begin{cases} |V(G^1_D)| < |V(G^2_D)|, & \text{or} \\ |V(G^1_D)| = |V(G^2_D)| \wedge |E(G^1_D)| < |E(G^2_D)|. \end{cases}$$

If, for a run system G^*_D representing D, there is no other run system G'_D representing D such that $G'_D <_r G^*_D$, we say that G^*_D is a *minimal* run system for D. ⌟

Removing nodes and edges from run systems which cannot be reached on a path beginning at a node of the form $(s, 0)$ is one way of making the run systems smaller according to the ordering defined above. Yet consider Figures 4.3 and 4.5. We gave Figure 4.3 as an example of what we were aiming for in producing a run system for the simple domain DS, but if that is correct then in generating the system depicted in Figure 4.5 we have missed our target, and in a way that the removal of unreachable nodes and edges cannot remedy. This raises the question of whether there are other rules we can give on how to reduce the graphs.

We proceed intuitively. Reduction might be thought to involve identifying vertices which were previously different, 'collapsing' a set of vertices onto a single representative of that set. Clearly, for any two vertices (s, n) and (s', n') in a set which is reducible, we must have $s = s'$. Assume G_D is a run system for the $\mathcal{C}+_{timed}$ domain D, and that there is $S \subseteq V(G_D)$, such that

- for all (s, n), (s', n') in S, we have $s = s'$;
- if $(s_0, n_0) \in S$ and $((s_0, n_0), e, (s_1, n_1)) \in E(G_D)$, then for all $(s'_0, n'_0) \in S$, we have $((s'_0, n'_0), e, (s_1, n_1)) \in E(G_D)$.

Then we might collapse the members of S onto a single representative, as follows. If $(s, 0) \in S$—an initial state—then we choose that vertex as the representative, otherwise we choose any member of S. Let the chosen member of S be (s^*, n^*). Then, we remove each edge $((s_0, n_0), e, (s, n))$, for $(s, n) \in S$, from G_D, and replace it by the edge $((s_0, n_0), e, (s^*, n^*))$. We also remove each edge $((s, n), e, (s_0, n_0))$, where $(s, n) \in S$, and replace it by $((s^*, n^*), e, (s_0, n_0))$. We then remove the vertices $S - \{(s^*, n^*)\}$ from G_D. This procedure is continued until there are no more sets S satisfying the properties given above.

Theorem 4.7 Let G^1_D represent D, and $S \subseteq V(G^1_D)$ satisfy the two constraints given above. Let G^2_D be the result of collapsing the members of S onto a single member (s^*, n^*) as described. Then there is a run

$$((s_0, 0), e_0, (s_1, n_1), \ldots, (s_t, n_t))$$

through G^1_D iff there is a run

$$((s_0, 0), e_0, (s_1, n'_1), \ldots, (s_t, n'_t))$$

through G^2_D.

Proof: Straightforward, by induction on the length t of runs. ⌟

Our notion of the reduction of a run system to one smaller but still representing the same action description of $\mathcal{C}+_{timed}$ is closely related to the modal-logical concept of a *bisimulation contraction*, and to work in automata theory on finding minimal representations for finite-state machines. A recent paper on efficient algorithms for computing bisimulation contractions is [DPP04], Section 2 of which provides a concise overview of work in this area.

Let us return to the action descriptions of Sections 4.3.4 and 4.3.5. It is clear by inspection that the subset comprising the two left-most nodes in Figure 4.5 satisfies the definition of S given above and can be collapsed, to the system shown in Figure 4.3. That run system is clearly minimal. And the system we generated for D^{pq}, and from which we removed the unreachable nodes and edges, can clearly be reduced to give the graph shown in Figure 4.7. That is

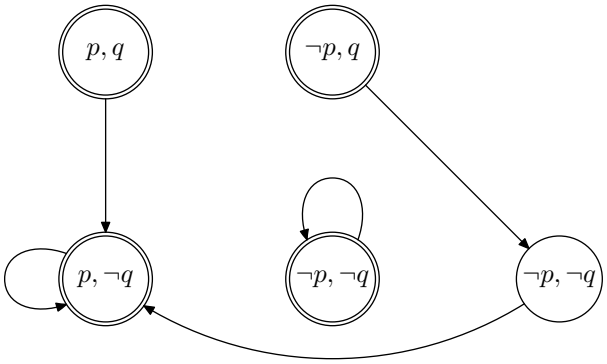

Figure 4.7: A minimal run system for D^{pq}

obvious because the two nodes

$$(\{p, \neg q\}, \{\mathbf{2}, \mathbf{1}\}) \qquad \text{and} \qquad (\{p, \neg q\}, \{\mathbf{2}\})$$

which have been collapsed onto a single node, have identical outgoing edges. Further reductions of this run system are not possible.

4.3.7 Third Example—Reagan and Gorbachev

As a third illustration, consider the following version of a familiar example from the field of deontic logic [Bel87]. (In deontic logic it is used in the study of 'contrary-to-duty' reasoning; we exploit it for different purposes here.)

If Reagan is told crucial strategic information, then Gorbachev must also be told, unless Gorbachev knows already; and vice versa. Once someone is told, then they know. We model this in $\mathcal{C}+_{timed}$ using a Boolean domain, with $\sigma^{smpl} = \{k_r, k_g\}$ and $\sigma^{ex} = \{r, g\}$. k_r represents that Reagan knows, and r is the action of telling Reagan; similarly for g and Gorbachev. The causal laws for this domain, D_{rg}:

inertial k_g,	k_g **if** $\lambda(g)$,
inertial k_r,	k_g **if** $\lambda(g)$,
g **if** $\neg k_g \wedge \lambda(r)$,	
r **if** $\neg k_r \wedge \lambda(g)$.	

Notice that the last two laws above could not have been expressed directly in $\mathcal{C}+$. Noteworthy is the fact that we only insist that Gorbachev (for example) should be told if an action of telling Reagan is performed; if the system starts in a state where there is disparity of knowledge, no telling need take place. A run system for this domain is shown in Figure 4.8.

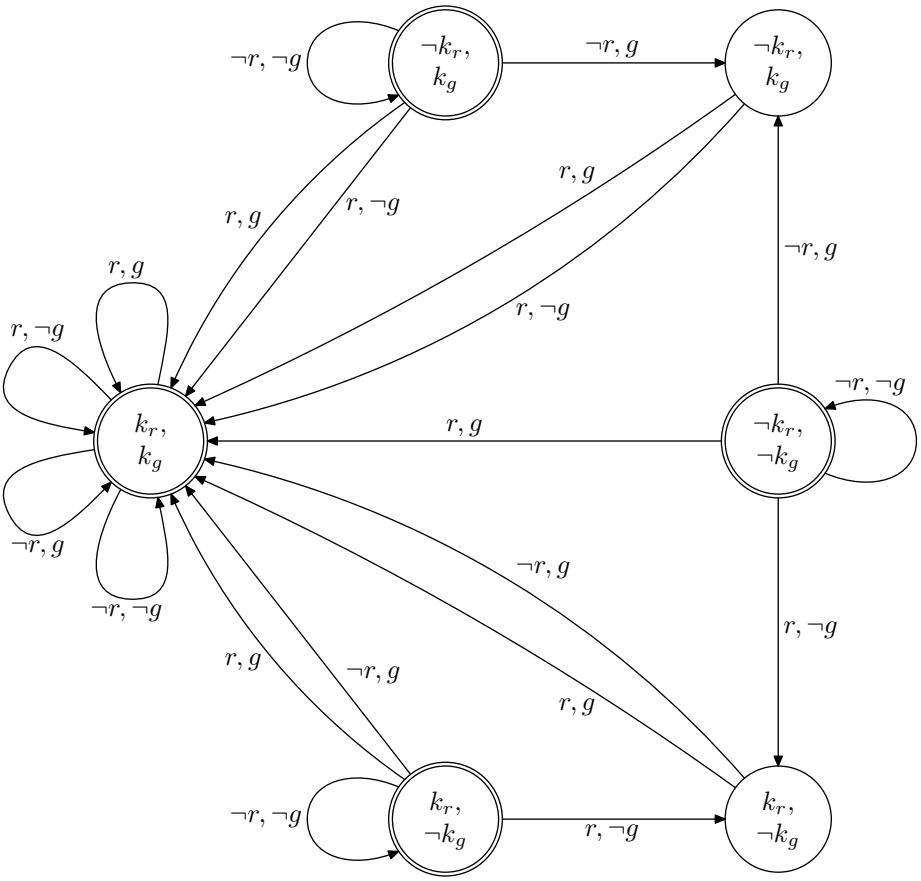

Figure 4.8: Reagan and Gorbachev

4.4 Interaction with $n\mathcal{C}+$

Say an agent promises to perform an action for another agent at some specified time in the future. If the agent fails to perform the promised action, we may wish to signal this as a breach of contract, regulation, or statement of assurance. This can be achieved using $n\mathcal{C}+$, which we described in Section 2.2.

However, the motivating example for $\mathcal{C}+_{timed}$ surfaces here in much the same form. For if the promise above is for an action to be performed 20 time-steps in the future, a proliferation of agents and promises would make our logical model complex and computationally inefficient. This argues in favour of a marriage of $n\mathcal{C}+$ and $\mathcal{C}+_{timed}$.

4.4.1 The Language $n\mathcal{C}+_{timed}$

We supplement the language of $\mathcal{C}+_{timed}$ with *permission laws*, which have the form

$$\textbf{not-permitted } F \textbf{ if } G. \tag{4.4}$$

Here, F is a formula, and G is a formula which can contain nested λ or $\top$, given as in (4.1). Further restrictions match those from Section 4.2: a permission law is simply a law of $\mathcal{C}+_{timed}$ with the prefix **not-permitted**. Where D is an action description of $n\mathcal{C}+_{timed}$, we say that the $\mathcal{C}+_{timed}$-*component* of D is that subset of its laws which do not contain the keyword **not-permitted**. Laws of $n\mathcal{C}+_{timed}$ can be given a canonical form just as for $\mathcal{C}+_{timed}$.

Translation to causal theories proceeds just as one would expect, given the translation for $n\mathcal{C}+$ described in [SC06] and presented in Section 2.2, and the translation of $\mathcal{C}+_{timed}$ domains into causal theories given earlier in the current chapter. Thus let D be an action description of $n\mathcal{C}+_{timed}$ with signature σ. The causal theory Γ_t^D consists of the causal rules

$$F[i+n_k] \Leftarrow \bigwedge_{j=0}^{k} G_j[i+(n_k-n_j)],$$

for each law of the form (4.2) in D and every $i \in \{0,\dots,t-n_k-1\}$, if F contains an action constant, and $i \in \{0,\dots,t-n_k\}$, otherwise; the rules

$$c[0]{=}v \Leftarrow c[0]{=}v$$

for each simple fluent constant c and $v \in dom(c)$; the rules

$$status[i+n_k]{=}red \Leftarrow F[i+n_k] \wedge \bigwedge_{j=0}^{k} G_j[i+(n_k-n_j)],$$

for each law (4.4) where $F \in fmla(\sigma^f)$ in D and $i \in \{0,\dots,t-n_k-1\}$; the rules

$$trans[i+n_k]{=}red \Leftarrow F[i+n_k] \wedge \bigwedge_{j=0}^{k} G_j[i+(n_k-n_j)],$$

for each permission law where F contains an action constant and $i \in \{0,\dots,t-n_k\}$; the rules

$$status[i]{=}green \Leftarrow status[i]{=}green$$

for $i \leqslant t$; the rules

$$trans[i]{=}green \Leftarrow trans[i]{=}green$$

for $i < t$; and finally, to ensure that a version of the *ggg* constraint is respected, the rules

$$trans[i]{=}red \Leftarrow status[i]{=}green \wedge status[i+1]{=}red$$

for each $i < t$.

Consider the example Boolean action description DP, whose signature is given by $\sigma^{smpl} = \{p\}$ and $\sigma^{a} = \{a\}$, and whose causal laws are

inertial p,
exogenous a
p **if** $\lambda(a)$,
not-permitted p **if** $\lambda^2(a)$.

What of the run systems? The means of generating these using causal theories and commitments, as described in Section 4.3.3, proceeds as before. This will give us the run system shown in Figure 4.9, for DP, after the graph G^*_D

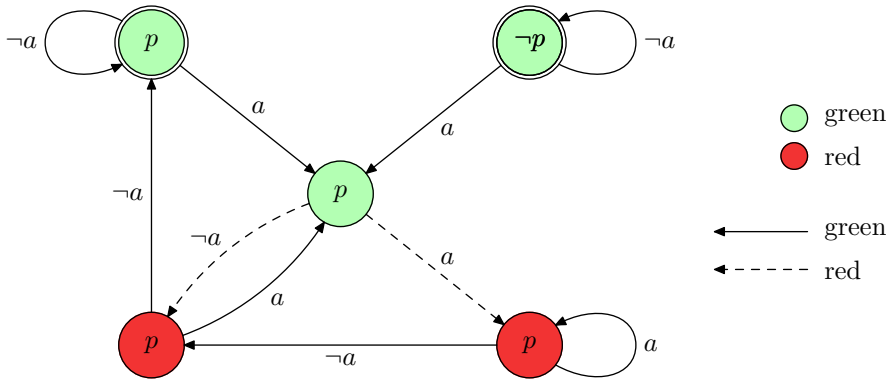

Figure 4.9: Run system for $n\mathcal{C}+_{timed}$ domain DP

has been fully reduced in accordance with the steps described in Section 4.3.6. This certainly corresponds to what had been intended in the action description.

Notice however that the graph does not simply represent a colouring of the run system defined by the $\mathcal{C}+_{timed}$-component of the action description DP. Thus we cannot say of $n\mathcal{C}+_{timed}$ that adding permission laws simply adds colour to the states and transitions—there may be a radical restructuring of the original run system, and in a certain sense this makes our models *deontically non-local*: the colour of states and transitions need not depend only on the interpretations (ignoring the values of the constants *status* and *trans*) which constitute those states and transitions. We can spell this out. Where x is a set of atoms, let $pure(x)$ be

$$x - \{c{=}v \mid (c = status \vee c = trans) \wedge v \in dom(c)\}.$$

Then, in our run system we may have vertices (s, n) and (s', n'), where $pure(s) = pure(s')$, but (s, n) and (s', n') are coloured differently. Similarly for transitions: it is possible to have two edges $((s_1, n_1), e_1, (s'_1, n'_1))$ and $((s_2, n_2), e_2, (s'_2, n'_2))$ where $pure(s_1) = pure(s_2)$, $pure(e_1) = pure(e_2)$ and $pure(s'_1) = pure(s'_2)$, but where the edges have different colours. We have both kinds of non-locality in the shown run-system for DP.

4.5 Summary

$\mathcal{C}+_{timed}$ is a natural generalization of $\mathcal{C}+$. Syntactically, $\mathcal{C}+_{timed}$ extends the laws of $\mathcal{C}+$ by adding a λ-operator allowing reference to past states and transitions: it thus removes the restriction to the immediately preceding state and transition. Semantically, it generalizes the transition systems defined by $\mathcal{C}+$ action descriptions. This new kind of transition system we have called a run system. The transition systems defined in $\mathcal{C}+$ are a special case where there is only one vertex for each state, and every vertex is initial. We have also shown how action descriptions of $\mathcal{C}+_{timed}$ are a shorthand for causal theories; using $\mathcal{C}+_{timed}$ we can make use of more of the language of causal theories, whilst retaining the key property: models of the causal theories are in correspondence with paths through the run system defined by the $\mathcal{C}+_{timed}$ action description. Computationally, action descriptions of $\mathcal{C}+_{timed}$ are much more efficient than attempted encodings of the same domains in $\mathcal{C}+$. The task of implementation is easy, through an adaptation of CCalc.

We have defined a means of generating a run system from a $\mathcal{C}+_{timed}$ action description. A further step of reduction is sometimes necessary to give the most compact run systems. Though the reduction steps are sound, we do not yet know whether they are complete, that is, whether they are guaranteed to result in run systems which are minimal according to the order introduced. Further work will explore this question. We are also interested in seeing whether the original generation of the run systems can be optimized so as to obviate reduction.

Chapter 5

$\mathcal{C}+$ and Model Checking

In the previous chapter we looked at increasing the expressivity of the causal laws which define an action description, augmenting the syntax of causal laws which can be written in $\mathcal{C}+$, to enable reference to states more than one time-step distant from each other, and actions at different time-steps. This made the representation of domains in which 'distant causation' occurs (such as those involving deadlines) much more convenient and perspicuous. In the present chapter, our focus will be on increasing the expressivity of query languages for $\mathcal{C}+$ action descriptions, rather than on the causal laws. The query language which currently exists for $\mathcal{C}+$ action descriptions, which was formalized in Section 2.1.8, is defined on runs, of finite length, through the transition system defined by an action description; we can ask only whether a set of specified atoms are true at given times. Yet the fact that $\mathcal{C}+$ laws define graphical structures of the sort which are very common in different branches of computer science, has suggested to us that we might use the technique of model-checking, frequently applied to these graphical structures, in order to verify whether properties hold which are not expressible in the standard query language. One cannot use current implementations of $\mathcal{C}+$ to ask whether some property of the system eventually holds, or to consider safety and 'liveness' properties of systems.

We have implemented three methods of connecting action descriptions of $\mathcal{C}+$ to model-checkers, two of which use NuSMV,[1] an industrially standard model checker which allows both Bounded Model Checking with SAT methods, and the older symbolic model checking using ordered binary decision diagrams (OBDDs). In this chapter we present the three implementations, together with examples and comparative studies of the performance of each. Finally, we discuss the possibilities of using each of our approaches to model-check $n\mathcal{C}+$, our deontic variant of $\mathcal{C}+$.

We will rely on the terminology and notation introduced in Section 2.5.1 in the following sections.

5.1 Interlude on FSMs

Before we proceed to give details on the various ways in which we have related $\mathcal{C}+$ to model-checking, a small technical point needs to be treated. As has

[1] Program and papers available from `http://nusmv.irst.itc.it/`.

been described, laws of $\mathcal{C}+$ define labelled transition systems, where the action constants σ^a of the signature of an action description are interpreted over the transitions, rather than at states. It is a substantial advantage of $\mathcal{C}+$ that actions, in this way, have first-class semantic citizenship. Yet the structures $M = (S, I, T, L)$ which are used by model-checkers have no content to their transitions, which are simply arrows between states: any atomic propositions must be interpreted on states, as the semantics we gave for LTL demonstrates. We cannot, at this stage, write about temporal logic specifications being true of systems defined in $\mathcal{C}+$, at least if we wish to allow action constants to occur in those specifications.

There is an easy remedy for this, well-known in computer science: in giving the graphical structure which an action description D of $\mathcal{C}+$ defines, we simply move the interpreted action constants (from the component e of a transition (s, e, s') in the transition system) back within the states of the FSM, and alter the arrows T between states accordingly. Reference to actions performed is then possible from the temporal logic defined over our structures, as the actions become atomic elements evaluated at states of the FSM.

Let us illustrate this with reference to the simple Boolean action description shown in Figure 5.1. There are three states; from two of these two actions

$\sigma^f = \{p, q\}$
$\sigma^a = \{a\}$

inertial p, q
exogenous a
q **if** p
p **if** $\top$ **after** a.
nonexecutable a **if** p

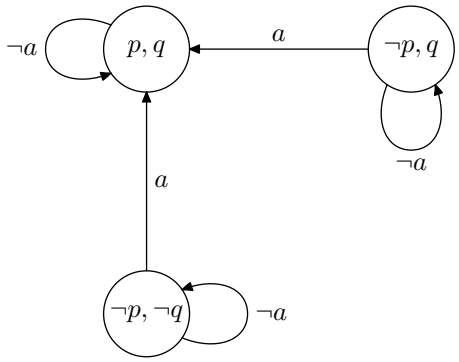

Figure 5.1: Simple action description and its transition system.

are possible ($\{a\}$ and $\{\neg a\}$), and from one state only one action is possible ($\{\neg a\}$, from the state where both p and q are true). This gives five possible pairings of actions with states, and so the FSM to which the action description of Figure 5.1 corresponds will have five states. It is depicted in Figure 5.2. Though runs through the transition system and runs through the FSM are not in one-to-one correspondence, it should be clear that there is a close relationship between the two structures. In fact, runs through the FSM map surjectively to runs through the transition system, as Theorem 5.2 shows.

In general, we derive the FSM directly from the transition system for an action description in the following way. Let D be an action description of $\mathcal{C}+$, with signature $\sigma = \sigma^f \cup \sigma^a$. We will say that for some interpretation s of σ^f, the *s-transitions* are those transitions (s, e, s') for some $e \in \mathrm{I}(\sigma^a)$ and $s' \in \mathrm{I}(\sigma^f)$. The *s-labels* are the members of $\mathrm{I}(\sigma^a)$ which feature as the second component of some s-transition.

Definition 5.1 The *FSM defined by the $\mathcal{C}+$ action description D* is the structure $M^D = (S^D, I^D, T^D, L^D)$, where the underlying set A of atoms is $\{c{=}v \mid c \in$

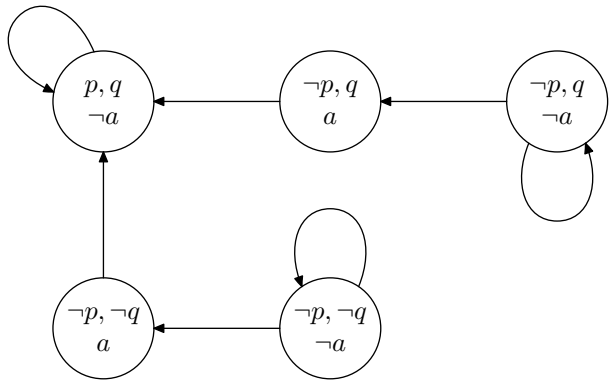

Figure 5.2: FSM for transition system in Figure 5.1.

$\sigma,\ v \in dom(c)\}$.

- S^D is the set of unions $s \cup e$, where s is a model of Γ_0^D and e is an s-label (if there are no s-labels, we include $s \cup \{\emptyset\}$);
- I^D is S^D (all states are possible initial states);
- $T^D \subseteq S^D \times S^D$ is such that $(s \cup e, s' \cup e') \in T^D$ (where $s, s' \in \mathrm{I}(\sigma^f)$) iff (s, e, s') is a transition of D—i.e. if $s[0] \cup e[0] \cup s'[1] \models_{\mathsf{C}} \Gamma_1^D$;
- $L^D : S^D \to \wp(A)$ is the identity (since states *are* sets of atoms).

A *run* (or *path*) through such a FSM is any (finite or infinite, ω-length) sequence of states $((s_0 \cup e_0), (s_1 \cup e_1), \ldots,)$, such that where the sequence is finite and terminates with $(s_n \cup e_n)$, for all i with $0 \leqslant i < n-1$, we have $((s_i \cup e_i), (s_{i+1} \cup e_{i+1})) \in T^D$. Where the run is infinite we should have $((s_i \cup e_i), (s_{i+1} \cup e_{i+1})) \in T^D$ for all i with $0 \leqslant i$. (We will usually write states as unions of the form $s \cup e$ for $s \in \mathrm{I}(\sigma^f)$ and $e \in \mathrm{I}(\sigma^a)$, as we have here, to facilitate the statement of relations to $\mathcal{C}+$.) ⌟

Given a run π (finite or infinite) through a FSM M^D defined by an action description D we let $\tau(\pi)$ be given by

- $\tau(\pi)$, where $\pi = ((s_0 \cup e_0), \ldots, (s_n \cup e_n))$ (π is finite) is
$$(s_0, e_0, s_1, e_1, \ldots, e_{n-1}, s_n),$$
- for infinite $\pi = ((s_0 \cup e_0), (s_1 \cup e_1), \ldots)$, $\tau(\pi)$ is
$$(s_0, e_0, s_1, e_1, \ldots)$$

As might be expected, we have the following easy correspondence theorem.

Theorem 5.2 Let D be an action description of $\mathcal{C}+$, and M^D the finite state machine defined by D. Then

- τ is injective from the set of infinite paths through M^D to the set of infinite runs through the transition system defined by D; and

- τ is surjective from the set of finite runs of M^D to the finite runs of D's transition system.

Proof: Let π be a run through the finite state machine, finite or infinite, as described. Clearly $\tau(\pi)$ is a run through the transition system immediately by definition. (For all i with $0 \leqslant i < n-1$, (s_i, e_i, s_{i+1}) is a transition of D.)
If π is infinite the proof of the specified injectivity is straightforward.
Suppose, on the other hand, that π is finite. For surjectivity, let $(s_0, e_0, \ldots, s_n)$ be a run through the transition system of D. Then if there are no s_n-labels, $\pi = (s_0 \cup e_0, \ldots s_n \cup \{\emptyset\})$ is a run through M^D with $\tau(\pi) = (s_0, e_0, \ldots, s_n)$. If there is at least one s_n-label, say e_n, then $(s_0 \cup e_0, \ldots s_n \cup e_n)$ is a run through M^D which corresponds by τ to $(s_0, e_0, \ldots, s_n)$. ⌟

(It is indeed clear that this correspondence is not one-to-one, as can be seen by considering our running example. For the runs in Figure 5.3, which start at the

Figure 5.3: Two runs through Figure 5.2

state depicted at the top-right in Figure 5.2, map to the same transition in the labelled system shown in Figure 5.1.)

The implementations we will present in the succeeding sections will be shown to produce, using a $\mathcal{C}+$ action description as raw material, a FSM as defined as above (or one which may be treated as though it were identical to that FSM). We now move on to the first of those implementations.

5.2 First Implementation

Our first method of connecting $\mathcal{C}+$ to model-checking supplements predicates which already exist in CCalc, without using an external model-checker such as NuSMV (as we will do with the second and third implementations).

Recall from Section 2.5.1 that the verification that a system satisfies a property is expressed in Bounded Model Checking (BMC) as the problem of finding models for the formula $[\![M, f]\!]_k$ defined as

$$[\![M]\!]_k \wedge \Big(\big(\neg L_k \wedge [\![f]\!]_k^0\big) \vee \bigvee_{l=0}^{k} \big(T(s_k, s_l) \wedge {}_l[\![f]\!]_k^0\big)\Big),$$

where $[\![M]\!]_k$ is satisfied by interpretations which represent runs through the transition system which represents the behaviour of the domain we are modelling. Yet we know that CCalc itself already produces a propositional representation of $[\![M]\!]_k$: this is simply the literal completion of Γ_k^D cast into conjunctive normal form (CNF), ready to be conjoined with a query to be passed to an external SAT-solver. Thus, in implementing BMC in CCalc, all we have left to do in

order to express $[\![M, f]\!]_k$ in CNF is to find a clausal equivalent of

$$\Big(\big(\neg L_k \wedge [\![f]\!]^0_k\big) \vee \bigvee_{l=0}^{k} \big(T(s_k, s_l) \wedge {}_l[\![f]\!]^0_k\big)\Big), \tag{5.1}$$

where f, as usual, is the negation of a specification in LTL. The result can then be conjoined to the clauses for $[\![M]\!]_k$ and sent for solution.

The left-hand disjunct of (5.1), as we saw in Section 2.5.1, represents the case where the run does not loop back to an earlier state of itself; that part of the formula is more straightforward to translate into CNF. The right-hand disjunct represents the case where there is a loop back from the k^{th} state to some earlier (the l^{th}) state, and this part requires a little more work.

The details are as follows. For the components $[\![f]\!]^0_k$ and ${}_l[\![f]\!]^0_k$ it closely matches the translation schemes given in Section 2.5.1, with modifications to ensure that the predicates do not loop infinitely in an attempt to produce a formula of infinitary logic. CCALC represents atoms of the signature of Γ^D_k by integers, and so the atomic clauses which define the translation of the formulas $[\![f]\!]^0_k$ and ${}_l[\![f]\!]^0_k$ will depend on a function ι whose domain is the set of atoms of Γ^D_k and whose range is the integers used by CCALC to represent those integers; this function encodes the CCALC representation. All we need to do is change the propositional and negation cases from the translation schemes given in Section 2.5.1. The updated clauses for $[\![f]\!]^0_k$ and ${}_l[\![f]\!]^0_k$ are the same:

$$[\![c{=}v]\!]^i_k := \iota(c[i]{=}v) \qquad {}_l[\![c{=}v]\!]^i_k := \iota(c[i]{=}v)$$
$$[\![\neg(c{=}v)]\!]^i_k := \neg\iota(c[i]{=}v) \qquad {}_l[\![\neg(c{=}v)]\!]^i_k := \neg\iota(c[i]{=}v)$$

Thus, suppose a very simple case in which the negation of the specification we wish to check is $\mathbf{X}(\mathit{loc}{=}\mathit{barn} \wedge \mathbf{X}\ \mathit{loc}{=}\mathit{barn})$, and we are checking with runs of length 3. We have

$$\begin{aligned}
[\![\mathbf{X}(\mathit{loc}{=}\mathit{barn} \wedge \mathbf{X}\ \mathit{loc}{=}\mathit{barn})]\!]^0_3 &= [\![\mathit{loc}{=}\mathit{barn} \wedge \mathbf{X}\ \mathit{loc}{=}\mathit{barn})]\!]^1_3 \\
&= [\![\mathit{loc}{=}\mathit{barn}]\!]^1_3 \wedge [\![\mathbf{X}\ \mathit{loc}{=}\mathit{barn})]\!]^1_3 \\
&= \iota(\mathit{loc}[1]{=}\mathit{barn}) \wedge [\![\mathit{loc}{=}\mathit{barn}]\!]^2_3 \\
&= \iota(\mathit{loc}[1]{=}\mathit{barn}) \wedge \iota(\mathit{loc}[2]{=}\mathit{barn})
\end{aligned}$$

The result of this translation process could be sent to a SAT-solver as it is, already in conjunctive normal form.

The causal theory Γ^D_1 defines the transitions of the labelled transition system of D; interpretations of the relevant signature which make the completion of Γ^D_1 true may be thought of as depicting transitions between states s_0 and s_1. In order to represent the formula $T(s_k, s_l)$, which is true when there is a transition from state s_k to state s_l of a run, we use variants of the propositional clauses depicting the completion of Γ^D_1: each fluent or action constant $c[0]$ occurring in the original clauses is replaced by a constant $c[k]$ (as the initial state of the transition is the k^{th}), and constants $c[1]$ are replaced by $c[l]$ (the transition moves to the l^{th} state).

Since, to cope with the case where there is a loop from the final state of the run s_k to some other state s_l, we must consider actions performed when the system is in s_k, additional action atoms will need to be added to the signature of

Γ_k^D. Normally, given an action description D in $\mathcal{C}+$, the signature of the causal theory Γ_k^D is made by time-stamping fluent constants with all t for $0 \leqslant t \leqslant k$, and all action constants with t for $0 \leqslant t < k$, the discrepancy of treatment reflecting the fact that runs are conceived of as ending with the last state, in which no actions are performed. Yet if there is a loop from s_k to s_l, there must be an action which effects that loop, and so we will need the members of σ^a stamped for time k; it is easy to define a procedure which adds these atoms to the count.

CCalc represents atoms of the signature of Γ_k^D by integers; we will introduce several new integers to enable our translation of $[\![M, f]\!]_k$ to be expressed more concisely. For complex action descriptions D the list of clauses representing the completion of Γ_1^D will be long, and since the clauses representing $T(s_k, s_l)$ have, as has been explained, the same structure as the completion of Γ_1^D, $T(s_k, s_l)$ will also be large. So, as $T(s_k, s_l)$, for each specific value of k and l, will turn out to appear many times in the propositional representation of $[\![M, f]\!]_k$ once this has been converted to CNF using De Morgan's laws, we will introduce integers ${}_l l_k$ representing each expansion of $T(s_k, s_l)$, using those integers in $[\![M, f]\!]_k$ instead, and adding to the clauses sent to the SAT-solver a representation of

$$ {}_l l_k \leftrightarrow T(s_k, s_l). $$

Such techniques are common in translating formulas to propositional logic, and repeated experimentation has shown their usefulness in the current context. We apply the same technique to the formulas ${}_l[\![f]\!]_k^0$ (for all l with $0 \leqslant l < k$) and $[\![f]\!]_k^0$, adding integers to represent each of these formulas, and using these single integers in place of the clauses throughout our translation.

We have written procedures implementing these methods for supplementing CCalc's own representation of $[\![M]\!]_k$ with a translation of

$$ \Big(\big(\neg L_k \wedge [\![f]\!]_k^0 \big) \vee \bigvee_{l=0}^{k} \big(T(s_k, s_l) \wedge {}_l[\![f]\!]_k^0 \big) \Big). $$

The implementation is in Prolog, and runs successfully alongside CCalc so that one may easily switch from performing bounded model-checking of LTL formulas to making standard queries in CCalc's own query language (that defined in Section 2.1.8). We will comment more on the performance and success of our first method in Section 5.5, where we compare our three implementations.

5.3 Second Implementation

As mentioned in the introduction to the current chapter, NuSMV is a state-of-the-art model-checker, used extensively both in research applications and industry. NuSMV works by constructing a representation of a finite state machine in OBDDs, or runs through a finite state machine in formulas of propositional logic, and adding to this a representation of a specification in a temporal logic. The user defines the finite state machine embodying the system's behaviour by writing a program in SMV.

An SMV program has several parts. The first is a specification `VAR` of a signature: a number of *state variables*, and a domain of values for each. Interpretations of the state variables determine the state of the system, so that

state variables are analogous to the fluent constants of $\mathcal{C}+$, with the exception that, as no formulas are evaluated on the edges of the finite state machine, any actions which are performed must, as has been described in Section 5.1, also be represented by SMV state variables.

SMV programs also contain an `ASSIGN` declaration: rules which specify the value of state variables at the next state, depending on what else is true at the next state, and what is true at the current state. (The similarity of these to fluent dynamic laws in $\mathcal{C}+$ is something we shall exploit in the current, second implementation.) The `ASSIGN` declaration also allows one to specify values which state variables must take initially.

An SMV program may also contain a `DEFINE` declaration. This allows one to introduce new state variables which do not have full semantic status, in that their values depend solely on the values of other state variables, and they are not used in the construction of NuSMV's internal representation of the model. They should be thought of as a convenient interface to the real state variables. We will not use a `DEFINE` declaration in our second implementation, but it is crucial to the utility of our third implementation.

Instead of defining a finite state machine implicitly, by many rules within an `ASSIGN` declaration which must be composed correctly to yield their semantics, one may include a `TRANS` declaration. This is a formula of propositional logic, written in terms of the current and next-state values of state variables. The transition relation of the finite state machine is then determined by the models of this formula. The `TRANS` declaration affords a very convenient, sometimes concise way of specifying the behaviour of a system, and we shall make use of it in our third implementation, where it is especially useful because we use very few–two, in fact—state variables.

Finally, SMV programs can contain a number of specifications in LTL or CTL, to verify upon the model defined by the rest of the code.

The second implementation constructs an SMV program which defines a FSM representing the behaviour of the system whose action description in $\mathcal{C}+$ is taken as input. The objective here has been to find close equivalents of the action description on a law-by-law basis, so that one can easily determine the parts of the SMV code whose inclusion is forced by the presence of a particular causal law. It ought to be possible, in many cases, for a human user of our implementation to look at the SMV code which is produced, understand it, and then to modify it. Our hope has also been that in basing our translations to SMV explicitly on the structure of laws rather than encoding the entire transition system, we will produce a more efficient representation of the system which we are modelling; this hypothesis will be investigated experimentally later in the chapter.

Thus, the signature of our $\mathcal{C}+$ action descriptions will map to a `VAR` declaration, giving the state variables in SMV. Causal laws of the action description will, very roughly, correspond to the clauses and cases of an `ASSIGN` declaration, though there are a number of complications here which will examine in the sections that follow. Finally, any queries found with the CCalc representation of the action description are translated to LTL.

5.3.1 Limitations

The second algorithm works only for a restricted subset of $\mathcal{C}+$ action descriptions. We here delineate that subset, and explain why the limitations are necessary.

The first limitation insists that the action descriptions we use should be definite. As it is rare, in our experience, that we encounter domains which must be formulated as non-definite action descriptions, this does not count as a substantial restriction for us. The reason for this narrowing of the classes of action descriptions is that SMV specifies the values for state variables on a variable-by-variable basis; because state variables will correspond to the constants of the signature of an action description, we cannot allow the heads of our causal laws to be anything other than an atom or $\bot$. (We could have allowed conjunctions of atoms in the heads of our causal laws, but in $\mathcal{C}+$ the transition system defined by an action description containing

$$c{=}v \wedge F \textbf{ if } G$$

(where G is either a formula of σ or else an expression G_1 **after** H befitting a fluent dynamic law) is the same as the transition system defined by the action description formed by replacing that causal law with the two others

$$c{=}v \textbf{ if } G \qquad \text{and} \qquad F \textbf{ if } G.$$

In this way, disallowing conjunctions in the heads of causal laws is not a real restriction.)

The second limitation is closely related to the concept of *dependence* which was brought forward to help specify $\mathcal{EC}+$ in Section 3.1: we alter this concept, applying it now to causal theories, though in much the same spirit as previously. Consider a causal theory Γ with signature σ. A *dependency graph* for Γ can be drawn, as follows:

- the nodes of the graph are the constants of σ;
- a (directed) edge goes from c to c' iff there is a causal rule in Γ with c' in the head and c in the body.

This is a very familiar idea in computer science. We will say that an action description D of $\mathcal{C}+$ exhibits *generalized dependency* when there is a circuit in the dependency graph of Γ_1^D which includes no edges from a node to itself.

The model-checker NuSMV, which takes as input specifications of systems written in SMV, will not accept definitions of the behaviour of the system in which there is a circular dependency among state variables. The way in which our second algorithm finds equivalents in SMV for $\mathcal{C}+$ action descriptions means that such circular dependency in the `ASSIGN` declaration of the source file we produce occurs when the $\mathcal{C}+$ action description exhibits generalized dependency, as defined above. Thus, if we wish to use NuSMV to verify properties of our systems, we must restrict the action descriptions we work with to those which do not have generalized dependency.[2] (We will remove these restrictions on the form of action descriptions in our third implementation.)

[2] Care has been taken here to refer to the acceptability of circular dependency to NuSMV, rather than the well-definedness of the semantics of circularly dependent SMV programs. As McMillan writes in his Ph.D. thesis: "In fact, this semantics assigns meaning to some programs which are not actually accepted by the compiler due to the rules regarding [...] circular dependencies" [McM93], page 118.

The third and final requirement on the form of $\mathcal{C}+$ action descriptions is more complicated, and relates to default conditions in causal laws and the causal rules derived from them. It is easiest to explain the restriction we are about to impose on the form of action descriptions by reference to restrictions on the SMV programs to which they correspond, and then to work backwards to the originating $\mathcal{C}+$ formulation.

When writing rules which determine the value of a state variable in SMV after a transition (rules which specify, for some state variable `c`, the value of `next(c)`) it is possible to be more or less comprehensive. Suppose that c can take three values, x, y, and z, and that there is another state variable d with the same domain. Rules such as

```
next(c) :=
  case
    c=x:     y;
    c=y:     z;
    c=z:     {x,y};
  esac;
```

or

```
next(c) :=
  case
    d=x:     x;
    1:       y;
  esac;
```

are exhaustive in their specification of what the value of c must be following a transition. Both groups of `case` expressions cover all possible cases: at least one of the expressions *must*, in both instances, be true. For the first series of expressions, this is because the value of c in the previous state must be one of x, y or z: these are the only possible values it can take; for the second rule, this is because, if $d{=}x$ is not true in the previous state, then there is a default condition which then constrains the value of c in the next state to be y. This means that the model-checker knows how to calculate the next value of c, based on information it already knows (even though, according to the first rule, when $c{=}z$ is true in the previous state, it must choose non-deterministically between x and y for the updated value).

If this sort of exhaustive coverage of cases does not occur, then NuSMV assumes, in the absence of an explicit default such as that in the right-hand rule, that the default value of the state variable is 1; here lies our problem. Since, to state the function of our algorithm crudely, conditions in these `case` expressions will roughly correspond to causal rules, NuSMV's assumption of a default value in the situation where, through lack of exhaustive coverage, it assumes one is needed, will be tantamount to the presumption of an additional causal rule in Γ_1^D—of the form

$$c[n]{=}\mathbf{t} \Leftarrow c[n]{=}\mathbf{t}.$$

Of course, this may disastrously affect the transition system which Γ_1^D is called on to define: for although we have that for all causal theories Γ, $\mathit{models}(\Gamma) \subseteq \mathit{models}(\Gamma \cup \{F \Leftarrow F\})$ (Proposition 14 of [SC05a]), this inclusion cannot be

strengthened to an equality, as discussion in [SC05a] demonstrates. Even worse, the value **t** may not even be in the domain of c.

To prevent these difficulties, we limit the set of $\mathcal{C}+$ action descriptions D to which our algorithm applies. Where c is a *simple fluent constant* of the signature of D:

- D should contain **inertial** c; or
- D should contain, for some $v \in dom(c)$, **default** $c{=}v$; or
- D should contain laws

$$\begin{array}{c} \textbf{caused } c{=}v_1 \textbf{ if } c'{=}v'_1, \\ \vdots \\ \textbf{caused } c{=}v_m \textbf{ if } c'{=}v'_n \end{array}$$

  where $dom(c') = \{v'_1, \ldots, v'_n\}$, or
- D should contain laws

$$\begin{array}{c} \textbf{caused } c{=}v_1 \textbf{ if } \top \textbf{ after } c'{=}v'_1, \\ \vdots \\ \textbf{caused } c{=}v_m \textbf{ if } \top \textbf{ after } c'{=}v'_n \end{array}$$

  where $dom(c') = \{v'_1, \ldots, v'_n\}$.

(The 'or' here is inclusive.) Where c is a *statically determined fluent constant*, we insist that the second or third of the above conditions should be satisfied—since where c is statically determined, laws of the first and fourth cases cannot exist. Finally, where c is an action constant, we require that

- **exogenous** c should be contained in D; or
- there should be action dynamic laws

$$\begin{array}{c} \textbf{caused } c{=}v_1 \textbf{ if } c'{=}v'_1, \\ \vdots \\ \textbf{caused } c{=}v_m \textbf{ if } c'{=}v'_n \end{array}$$

  where $dom(c') = \{v'_1, \ldots, v'_n\}$.

It should be clear what effect the satisfaction of these constraints will have on the causal theories Γ_0^D and Γ_1^D, which determine the states and transitions of the labelled transition system underlying D. Any time-stamped constant $c[0]$ in the signature of Γ_0^D, or constant $c[0]$ or $c[1]$ in the signature of Γ_1^D, will be caused to have a value in the reduct $(\Gamma_n^D)^X$ ($n \in \{0, 1\}$), regardless of the interpretation X. Further, the SMV `case` expressions which are produced by our algorithm will have that kind of exhaustive coverage of cases which stops NuSMV from providing implicit default values of its own when interpreting the rules for initial and next-state values of those state variables to which the constants of D will correspond.

It is frustrating to have to make this third restriction. In our experience, the second narrowing of the class of action descriptions which we imposed—the exclusion of laws which have generalized dependency loops—has affected very few of the action descriptions with which we are concerned. In fact, to our knowledge, the only action descriptions which have been excluded under these rules are invented domains not formalized from real-world examples. The case with the third restriction is different: though it is not the norm, there are action descriptions which do not conform to the rules we gave above. We cannot apply this second algorithm to these action descriptions. (This limitation was a principal factor in motivating the development of our third algorithm for linking model-checking to action languages; this is discussed below, in Section 5.5.)

5.3.2 Details of the Second Approach

It is easy to check whether an action description of $\mathcal{C}+$ conforms to the three restrictions made above. The first restriction of definiteness is immediately visible, and the third restriction is also easy to verify. The second rule we imposed, that our causal laws should not exhibit what we called *generalized dependency*, is slightly more time-consuming as it involved traversing the dependency graph which we described in the previous section. Nevertheless, all three checks can be performed automatically and quickly.

Given an action description of $\mathcal{C}+$ conforming to the three restrictions above, we first form a list of all causal rules which are members of the causal theory Γ_1^D (recall that the models of this causal theory are precisely the transitions of the labelled transition system defined by the $\mathcal{C}+$ action description). Since CCalc itself converts the $\mathcal{C}+$ action description into Γ_1^D when the file is loaded, this is simply a matter of extracting the causal rules from the database of a Prolog system running CCalc. The predicate which does this is `mc2_get_causal_rules/1`, and the rules themselves are after bound to the variable `Rules`. It is from these causal rules (rather than from the causal laws which were used to generate them) that we will make our SMV program.

Note that there may be causal rules in Γ_1^D which have as their head $\bot$; if there are any such rules, then we record that fact. Such causal rules stem from causal laws of $\mathcal{C}+$ which express that a given combination of fluent atoms cannot occur in a state, or that some given action is not possible when the system described is in such-and-such a state. Thus, though a formal argument is a little more complex, these laws can only *remove* states and transitions from the transition system defined by the original action description of $\mathcal{C}+$. There will be special treatment in our algorithm for the causal rules which derive from such laws, and the treatment will differ depending on whether the causal rule stems from a static or action dynamic law on the one hand, or a fluent dynamic law on the other, in the original action description. The reason that privileged treatment is necessary here is that it is often awkward in SMV to specify that certain combinations of values for state variables do not occur, or that, under given conditions, a transition between states of the system is impossible. There will be more discussion of the details of this special treatment below.

After the causal rules have been found, the program invokes the predicate `mc2_get_constants/1` to find an ordered list of all the (fluent and action) constants of the signature of D; the list is bound to `Constants`.

Now the first phase of writing the external SMV file begins, which is to specify the signature for the SMV program. This mirrors very closely the signature σ of the $\mathcal{C}+$ action description, the differences being forced by constraints imposed on the syntax of state variables and their values in SMV. We loop over `Constants`, and for each constant c of σ include a declaration of its presence and the values it takes in the output file. Brackets in the strings representing c and its values are replaced by underscores. For instance, the signature of our simple action description depicted in Figure 5.1, where $\sigma^f = \{p, q\}$ and $\sigma^a = \{a\}$, and all constants are Boolean, determines the following SMV code:

```
MODULE main
  VAR
    a:                  boolean;
    p:                  boolean;
    q:                  boolean;
```

If we had included another action constant *walk(hagar)* (to represent the action of Hagar's walking), whose values were {*quickly*, *slowly*}, then the line

```
    walk_hagar:         {quickly,slowly};
```

would also occur in the output file; this illustrates the treatment of brackets.

At this stage we also check whether any causal rules with $\bot$ were found earlier as members of Γ_1^D, and if this is so, we include a special Boolean state variable `false` in the SMV code:

```
    false:              boolean;
```

By default, this state variable is evaluated as false, yet we will later specify that the conditions under which it is true are precisely those which satisfy the body of a causal rule in Γ_1^D whose head is $\bot$. Thus when model-checking using SMV code generated by the current procedure, for action descriptions which have laws with $\bot$ in the head, we will restrict attention to runs of the finite state machine along which `false` is never true. This is easy to do: given an LTL or CTL specification F, instead of checking for $\neg F$ we simply check for

$$(\mathbf{G}\ \neg\texttt{false}) \rightarrow \neg F$$

instead. In this way we ensure that states and transitions ruled out of the transition system by causal laws of the original action description with the head $\bot$, do not feature in the finite state machines defined in SMV.

As an example, consider again the action description shown in Figure 5.1. This contains the causal law

$$\textbf{nonexecutable}\ a\ \textbf{if}\ p,$$

which means that the causal rule

$$\bot \Leftarrow a[0] \wedge p[0]$$

is a member of Γ_1^D. The finite state machine defined by the SMV code which our second algorithm produces is shown in Figure 5.4; the drawing conventions here are that those states in which `false` is evaluated as true (in SMV, assigned value

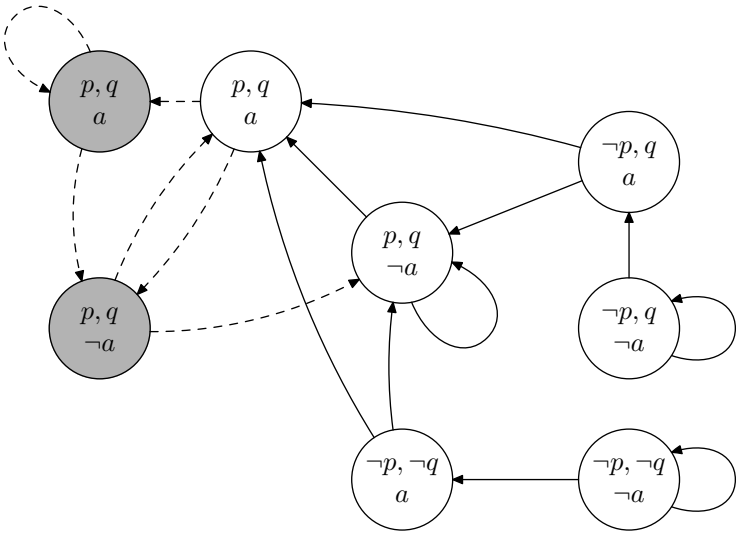

Figure 5.4: FSM actually defined by SMV code for domain of Figure 5.1.

1) are shaded, and that transitions linking to or from such states are dashed rather than drawn continuously. The subgraph of that finite state machine comprising states in which `false` is true (and from which we remove transitions connected to states which make `false` true) is clearly the same as the FSM given in Figure 5.2, which is obtained directly from the labelled transition system which the $\mathcal{C}+$ action description determines. Thus in this instance, restricting the model-checker to runs along which the state variable `false` is false will ensure that counterexamples to any specifications presented for verification are indeed counterexamples, rather than spurious runs which should not be possible given the presence of the relevant law, or laws, with $\bot$ in the head.

So much for the specification of state variables and related matters. We turn back to the stages of our algorithm: the next is to write the `ASSIGN` declaration to the SMV file, which specifies what the initial values of state variables may be, and also how their values in the state after a transition depend on their values before the transition is made. The outermost loop for this stage is controlled by the predicate `mc2_assign_loop_aux/3`, which recurses over the list of constants `Constants` which we bound earlier. This is the most involved part of the algorithm, for we here seek to extract SMV conditions governing the behaviour of individual state constants from the causal theory into which CCalc has converted the original $\mathcal{C}+$ action description. There are two aspects of this stage which require particular care.

The first is how to deal with potential conflicts between causal laws. Consider the action description and transition system in Figure 5.5. In the situation where both p and q are true in a state, there are no outgoing transitions, as both a and $\neg a$ would be caused. Yet the structure and semantics for SMV are such that this is difficult to constrain, and we must make use of our privileged state variable `false`. We do this as follows. For any constant c of the $\mathcal{C}+$ action

$\sigma^f = \{p, q\}$
$\sigma^a = \{a\}$

inertial p, q
exogenous a
caused a **if** p
caused $\neg a$ **if** $p \wedge q$
caused p **if** $\top$

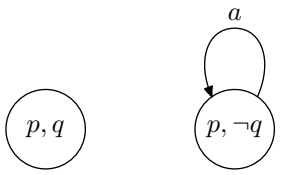

Figure 5.5: Conflicts in causal laws

description, we check to see whether there are causal rules

$$c[n]{=}v_1 \Leftarrow F_1 \qquad \text{and} \qquad c[n]{=}v_2 \Leftarrow F_2,$$

with $v_1 \neq v_2$, in Γ_1^D, such that F_1 and F_2 may be true together. Such checking may be performed more or less thoroughly: the simplest sound method is—supposing, as usual, that causal laws have been cast into a canonical form where the bodies are conjunctions of atoms—to check whether there is a constant c^* such that, for some $n^* \in \{0, 1\}$, $c^*[n^*]{=}v$ appears as a conjunct in F_1 and $c^*[n^*]{=}v'$ is a conjunct in F_2, for $v \neq v'$. If such mutually exclusive time-stamped atoms do *not* appear in the bodies of potentially conflicting causal rules, it may be the case that the bodies can be true together, and in this case we should include an appropriate condition in the `ASSIGN` declaration for `false`:

```
ASSIGN
  next(false) :=
    case
      F1 & F2:                  1;
      <other cases>
      1:                        0;
    esac;
```

We can then restrict model-checking to those runs along which `false` is forever false, as previously explained. Thus there are two situations in which a state variable `false` is included in the signature for our SMV program: first, where there is some causal rule

$$\bot \Leftarrow F$$

in the theory Γ_1^D; and second, where there is a potential conflict in the causal rules of that theory, as described above.

(As implied, there are more thorough methods of checking whether F_1 and F_2 can be true together, and thus whether a conflict may arise. For example, consider the case where there are the following static laws as part of an action description:

caused p **if** q
caused $\neg p$ **if** $q \wedge r$
caused $\neg r$ **if** $\top$.

The checking we have currently implemented would see the causal rules in Γ_1^D descended from the first two laws, identify them as potentially in conflict, and add a condition to the clauses governing `false` as follows:

```
next(false) :=
  case
    q & r:                    1;
    1:                        0;
  esac;
```

Yet a more thoroughgoing, intelligent, global analysis of Γ_1^D, even of a relatively simple kind, would have found that q and r can never be true together, as r must always be false (because of the presence of **caused** $\neg r$ **if** $\top$ in the action description). For the time being, we do not make our analysis more searching, in directions such as this—it being, in our experience, infrequent that conflicts arise in causal rules at all.)

We move on to the second feature of our `ASSIGN` declarations which deserves attention. An attractive feature of $\mathcal{C}+$ is its expressivity in regard to defaults and inertia: it is possible to turn inertia on and off for particular fluent constants (and, indeed, for particular values of fluent constants), to say that a given fluent is inertial only under given conditions, and to stipulate that there are multiple possible default values for a given fluent constant or action constant. Now, it sometimes arises that two different defaults are activated for the same constant. Consider the example depicted in Figure 5.6. For this system, Γ_1^D consists of

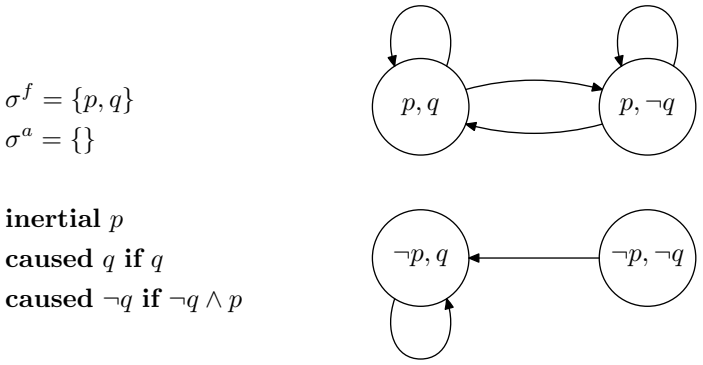

Figure 5.6: Action description with multiple defaults.

the causal rules:

$$\begin{aligned} p[1] &\Leftarrow p[1] \wedge p[0] \\ \neg p[1] &\Leftarrow \neg p[1] \wedge \neg p[0] \\ q[0] &\Leftarrow q[0] \\ \neg q[0] &\Leftarrow \neg q[0] \wedge p[0] \\ q[1] &\Leftarrow q[1] \\ \neg q[1] &\Leftarrow \neg q[1] \wedge p[1]. \end{aligned}$$

It is clearly the presence of the two last causal rules which makes the transition system (shown in Figure 5.6) non-deterministic: where p is true, q may be either true or false, and can change exogenously between these values.

In general, the phenomenon in which we are interested occurs when there are at least two rules

$$c[n]{=}v_1 \Leftarrow c[n]{=}v_1 \wedge F_1 \quad \text{and} \quad c[n]{=}v_2 \Leftarrow c[n]{=}v_2 \wedge F_2$$

in the causal theory Γ_1^D, with $v_1 \neq v_2$, and where the bodies F_1 and F_2 are not mutually exclusive (where this is defined as previously, for the case of conflicting causal rules without defaults).[3]

SMV evaluates conditions within any given `case` clause in strict, unvarying, top-down order. Thus in cases where there are different default conditions for a constant which may be activated simultaneously, we cannot use a single `case` expression to capture the behaviour of our system, since this would necessarily give priority to whichever of the default rules we wrote first, and runs where the value of the relevant constant was determined by one of the other default conditions would be overlooked.

Accordingly, we will make use of a union of multiple `case` expressions. Suppose that for some given constant c of an action description, there are n causal rules containing default conditions, as follows:

$$\begin{gathered} c[t]{=}v_1 \Leftarrow c[t]{=}v_1 \wedge F_1 \\ \vdots \\ c[t]{=}v_n \Leftarrow c[t]{=}v_n \wedge F_n. \end{gathered}$$

Let us further suppose that in Γ_1^D there are a number of rules with the same time-stamp t and constant c in their head, which are *not* defaults—i.e., where whichever time-stamped atom contained in the head does not also occur in the body of the causal rule. Let these other causal rules be

$$\begin{gathered} c[n]{=}v_1' \Leftarrow G_1, \\ \vdots \\ c[n]{=}v_m' \Leftarrow G_m. \end{gathered}$$

Then, in writing the `ASSIGN` expression which defines the behaviour of $c[n]$, we will have one `case` expression for each different ordering of the causal rules which contain default conditions; these `case` expression will then be linked together by `union`. This simulates soundly the non-determinism which results from multiple defaults in $\mathcal{C}+$.

By way of illustration, consider the action description in Figure 5.7. For the fluent constant p (with domain $\{x, y, z\}$) time-stamped for 1, there is one causal rule in Γ_1^D which is not a default, and three rules which are defaults. In the SMV code produced by the algorithm, there is one `case` clause for each possible ordering of the default rules: six clauses in all. Here is the code, governing the value of p in the next state, which our algorithm actually produces:

```
ASSIGN
  next(p) :=
    case
```

[3] This type of interaction between different defaults is the kind explicitly outlawed according to the restrictions we placed on $\mathcal{EC}+$ in Section 3.1.

$\sigma^f = \{p\} \quad (dom(p) = \{x, y, z\})$
$\sigma^a = \{a\}$

exogenous a
default $p{=}x$
default $p{=}y$
default $p{=}z$
caused $p{=}x$ **if** $\top$ **after** a

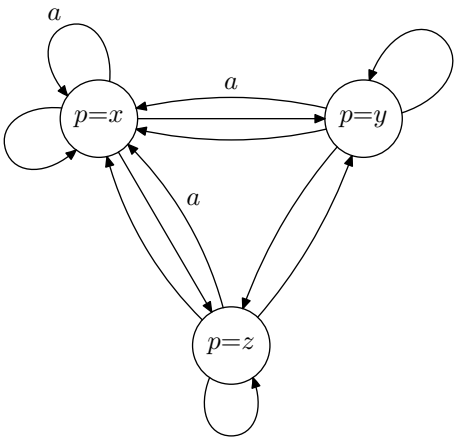

Figure 5.7: Another action description with multiple defaults.

```
  a: x;   1: z;   1: y;   1: x;
esac union
case
  a: x;   1: y;   1: z;   1: x;
esac union
case
  a: x;   1: y;   1: x;   1: z;
esac union
case
  a: x;   1: z;   1: x;   1: y;
esac union
case
  a: x;   1: x;   1: z;   1: y;
esac union
case
  a: x;   1: x;   1: y;   1: z;
esac;
```

In the current example, six clauses are not strictly necessary:[4] three would suffice. The values which p may take after a transition has been made are defined by the `ASSIGN` declaration for `next(p)`: p must always take a value in accordance with one of the `case` clauses (which thereby function disjunctively), and within a `case` clause, it takes the first value such that is paired with conditions true of the transition (this is standard for SMV, and will not further be explained).

With these two preliminary points concluded, the continuation of the description of our second algorithm which follows should be more easily understood.

As noted, the phase where we write the `ASSIGN` declaration is controlled by the predicate `mc2_assign_loop_aux/3`, which recurses over the list of constants `Constants` (this list may include `false` if the conditions for its inclusion hold).

[4]This is owing to the fact that, in all of the causal rules $p[1]{=}v \Leftarrow p[1]{=}v \wedge F$ which have default conditions for $p[1]$, the components F are identical (in fact, they are empty). Where the components F differ, all possible orderings of the rules may be needed.

Already bound is a list of the causal rules of Γ_1^D, to `Rules`.

For each constant c of the action description, we first remove from `Rules` those causal rules which govern the default and inertial behaviour of c. These are the rules which have the form

$$c[0]{=}v \Leftarrow c[0]{=}v \wedge F$$

or

$$c[1]{=}v \Leftarrow c[1]{=}v \wedge F$$

(assuming that the causal rules have been normalized so that their bodies are conjunctions of atoms or $\top$, and that an appropriate ordering has been placed on the conjuncts). The first kind of rule, with $c[0]{=}v$ in the head, is a default condition on the value of c; it has the effect of making c take the value v by default, when the additional conditions F are true. The second kind of rule may also state a default value of c (in the state succeeding a transition), or it may, if the atom $c[0]{=}v$ is one of the conjuncts in F, impose a condition of inertia on $c{=}v$ (again, supposing any *other* conjuncts in F are true). Inertia is thus a special kind of default: default persistence. The default rules with $c[0]{=}v$ in the head are bound to `DefRules0`, and those with $c[0]{=}v$ in the head to `DefRules1`. Now the SMV clauses determining the `init` and `next` values of c can be written—for both there is a union of `case` expressions, one for each permutation of the laws in `DefLaws0` (for `init`) and `DefLaws1` (for `next`). Within each `case` clause, the non-default rules from `Rules` for the constant c are written first, then the particular ordering of defaults.

5.3.3 Queries

When the `ASSIGN` declaration has been written, it remains to translate the queries which may have been passed, together with the $\mathcal{C}+$ action description, to our algorithm. These queries may be either in the standard query language of CCalc (see Section 2.1.8), in which case they have the form of a triple (L, T, N), with

- $L \in \mathbb{N}$ a unique identifier;
- T as $[t_{min}, t_{max}]$, where this denotes an interval of $\mathbb{N}$ and $t_{min} \leqslant t_{max}$, or T is $[t, \infty)$, $t \in \mathbb{N}$;
- A as a set of atoms $c[i]{=}v$, where $i \in T$, and if $c \in \sigma^a$ and $T = [t_{min}, t_{max}]$, then $i < t_{max}$; or else i is *max* and $c \in \sigma^f$;

or else they may be written in one of the temporal logics which NuSMV accepts as input specifications. If the query is of the latter type, it can straightforwardly be written to the SMV file, with the exception that when we have relied on the state variable `false` to determine the correct behaviour of the finite state machine, the query becomes

$$(\mathbf{G}\ \neg\texttt{false}) \rightarrow \neg F,$$

where F is the original formula of temporal logic.

If the query is in CCalc's standard query language, it needs to be translated into temporal logic: we use LTL as it affords an easy translation scheme. Thus

consider some query (L, T, N), and suppose that atoms with the same time stamp have been conjoined; thus N will have the form

$$\{F_1[t_1], \ldots, F_n[t_n], F_{max}[max]\}$$

with $t_i \in \mathbb{N}$ for all i such that $1 \leqslant i \leqslant n$. We will permit, as a shorthand, expressions $\mathbf{X}^n$ to denote n occurrences of the temporal operator $\mathbf{X}$: thus $(\mathbf{X}^2\ F)$ is true of a path $(s_0, s_1, \ldots)$ when F is true of $(s_2, s_3, \ldots)$. The query A above is accordingly translated into the formula F:

$$(\mathbf{X}^{t_1}\ F_1) \wedge \cdots \wedge (\mathbf{X}^{t_n}\ F_n) \wedge (\mathbf{X}^{t_{min}}\ (\mathbf{F}\ F_{max})).$$

The initial series of conjuncts here ensure that the formulas F_i, where i is a number and not *max*, are true at the correct distances along a run. For the formula F_{max} which must be true at the final state s_m of a run, we must ensure that $m \geqslant \max\{t_{min}, t_n\}$. m should be greater than or equal to t_{min} so that we respect the minimum bound on the length of runs which the user has supplied; and m must be greater than or equal to t_n to ensure that F_{max} is not satisfied somewhere in the middle of a run. Where the interval T of the original query is $[t_{min}, \infty)$, no further work needs to be done; if the interval is closed, then we can impose an upper bound on the length of runs to be considered by a command-line switch when operating NuSMV.

Of course, queries can be passed to NuSMV directly, expressed in any of the temporal logics which that model-checker accepts as input; in this case, care must be taken to ensure that where a state variable `false` has been included as part of the SMV clauses, NuSMV is directed to consider those runs through the FSM along which `false` is never true.

5.3.4 Remarks

All of the above has been implemented in PROLOG and, just as with the first method for model-checking with $\mathcal{C}+$, runs successfully alongside the normal operations of CCALC.

The details of the second approach which we have just given, in Section 5.3.2, are somewhat informal. Clearly it would be desirable to present the algorithm for finding equivalents, in SMV, of restricted $\mathcal{C}+$ action descriptions in more rigorous detail. Further, although the method is intuitively correct and has confirmed this intuition in all examples on which we have checked it, it would be much better to have a proof of the soundness of our approach, in the form of a theorem stating that runs of the action description correspond to runs (where `false` is not true) through the finite state machine of the SMV code. One reason we have been prevented from proving such a theorem is the inherent complexity of the algorithm for generating SMV code, which also results in very complex SMV programs. Yet perhaps the main reason is that the semantics of the SMV language are not very clearly defined in McMillan's Ph.D. thesis [McM93], where the language is introduced. In particular, not all of the constructs of the language we make use of in our files are given semantics, and so proving the correctness of our second algorithm would first involve an involved reconstruction of the meaning of SMV expressions.

The difficulty involved in achieving certain confidence in the correctness of this second implementation was one of the factors which prompted us to try

a third approach where, as we will see, knowledge of the correctness of the procedure is more forthcoming.

5.4 Third Implementation

The second way of adapting CCalc to the purposes of model-checking, presented in the previous sections, suffers from a significant limitation: it cannot cope, even in principle, with all classes of $\mathcal{C}+$ action description. As has been described, action descriptions whose atoms exhibit a dependency loop through static conditions cannot be translated to SMV under this scheme. In practice many of the action descriptions which we encounter and which we wish to reason about do not have this form of unfortunate dependency. We fare worse with the third kind of limitation, that relating to constants which are not caused to have a value in the causal theories Γ_0^D and Γ_1^D, as these forms of action description have been, in our experience, more common; we see no reason to suppose that our experience here has been unusual and unrepresentative. Thus there is a good reason for trying to find ways of sidestepping these obstacles, to look for alternative correlations between action descriptions and SMV programs—in addition to the independent interest of alternative implementations generally. Further, as has been said, we wished to make it easier to verify that a finite state machine encoding the same behaviour as the labelled transition system of $\mathcal{C}+$ is produced.

In this section we present the last of our approaches. It is similar to the second, in that it takes an action description in $\mathcal{C}+$, together with queries both in CCalc's standard query language or LTL, and outputs SMV code which can immediately be passed to NuSMV. However, this third implementation copes with all forms of action description. Broadly, it finds all states and transitions of the system, and then writes a representation of this information in SMV.

As has repeatedly been stated, the states of a transition system defined by a $\mathcal{C}+$ action description D are simply the models of Γ_0^D, and the transitions of the system are in one-to-one correspondence with the models of Γ_1^D, from which the transitions may be extracted very easily. For models of Γ_1^D have the form $s[0] \cup e[0] \cup s'[1]$ for some $s, s' \in \mathrm{I}(\sigma^f)$ and $e \in \sigma^a$, and (s, e, s') is, for such a model, a transition of the system. Thus, we may use CCalc to find all states and transitions for D by finding all models of, respectively, Γ_0^D and Γ_1^D.

The SMV code we produce will have two state variables, *state_id* and *action_id*. Let $\{s_0, \ldots, s_n\}$ be the set of states of an action description D. The state variable *state_id* will take values from the set of *atoms* $\{s_0, \ldots, s_n\}$ (we use the same expressions to denote both atoms of the SMV code and interpretations of $\mathrm{I}(\sigma^f)$). We will say that the *labels* of an action description D are the set

$$\{e \mid \exists s, s' \in \mathit{states}(D)\ ((s, e, s') \in \mathit{trans}(D))\}$$

so that the labels of D are the union of all the s-labels, for all states s of the transition system. If $\{a_0, \ldots, a_m\}$ is the set of labels of D, then the state variable *action_id* will take values from the set of *atoms* (same polysemy) $\{a_0, \ldots, a_m\}$. For example, the simple action description shown in Figure 5.1, when treated by the third model-checking algorithm, yields code which begins as follows:

```
MODULE main
```

```
VAR

  state_id:       {s0,s1,s2};
  action_id:      {a0,a1};
```

This is because there are three states in the action description—specifically, $\{p, q\}$, $\{\neg p, \neg q\}$ and $\{\neg p, q\}$—and two types of transition—being $\{a\}$ and $\{\neg a\}$.

We use a `DEFINE` declaration in the SMV program to encode the interpretation of each fluent and atom constant in each state or transition label. New variables are introduced corresponding to each constant of the signature σ of the $\mathcal{C}+$ action description, and the values of those constants are defined in terms of the value of *state_id* (for variables corresponding to fluent constants) and *action_id* (for variables corresponding to action constants). Here is the relevant declaration for the example of the last paragraph:

```
DEFINE

  p :=
    case
      state_id=s0:          1;
      1:                    0;
    esac;
  q :=
    case
      state_id=s0
        | state_id=s2:      1;
      1:                    0;
    esac;

  a :=
    case
      action_id=a1:         1;
      1:                    0;
    esac;
```

As can be seen, the `DEFINE` declaration has been divided into clauses, each of which governs the interpretation of a constant from the original signature σ of the $\mathcal{C}+$ action description. The fluent constants come first, in a block, followed by the action constants. The definitions in the block of SMV code make it clear that, in the current instance, the state $\{p, q\}$ is to be associated with the value s_0 of *state_id*, and $\{\neg p, \neg q\}$ and $\{\neg p, q\}$ with the values s_1 and s_2 respectively. The actions are treated in similar fashion. The utility of `DEFINE` declarations should be obvious: they enable us to make queries, to model-check, using the fluent and action constants of our original action description, rather than in terms of our new state variables *state_id* and *action_id*.

Owing to the conventions operant on the SMV language, we must sometimes make slight, inessential alterations in the form which our constants and their values take in these `DEFINE` declarations. Brackets are typically replaced by underscores, and some values for constants are replaced by numerals, as described in Section 5.3.2. Where substitutions of the latter type are made, they are noted in the SMV code, as comments, and also on the command line.

The next portion of the SMV program, after the variable declarations and further definitions of constants, is the representation of the transition system; this is mostly very straightforward. As we have included two state variables *state_id* and *action_id* in the code, we can associate the *next* value of *state_id* with combinations of the values of *state_id* and *action_id* which refer to the current state (of the FSM). Which associations and combinations to use can be gleaned directly from information about the models of Γ_1^D. In the simple example which we have been using, and which was depicted in Figure 5.1, the s-transitions for the various states of the system are as follows:

$$\begin{aligned}
\{p,q\}: &\quad (\{p,q\},\{\neg a\},\{p,q\})\\
\{\neg p,q\}: &\quad (\{\neg p,q\},\{\neg a\},\{\neg p,q\})\\
&\quad (\{\neg p,q\},\{a\},\{p,q\})\\
\{\neg p,\neg q\}: &\quad (\{\neg p,\neg q\},\{\neg a\},\{\neg p,\neg q\})\\
&\quad (\{\neg p,\neg q\},\{a\},\{p,q\})
\end{aligned}$$

Accordingly, the `TRANS` declaration is:

```
TRANS

  (state_id=s0 ->   ( action_id=a0 & next(state_id)=s0)) &
  (state_id=s1 ->   ( (action_id=a0 & next(state_id)=s1)
                    | (action_id=a1 & next(state_id)=s0))) &
  (state_id=s2 ->   ( (action_id=a0 & next(state_id)=s2)
                    | (action_id=a1 & next(state_id)=s0)))
```

For each member s of $states(D)$, we find all actions e and states s' such that $s[0] \cup e[0] \cup s'[1]$ is a model of Γ_1^D. If there are no such actions and states (the case where there are no outgoing transitions from state s), we include a clause

```
  (state_id=s  ->   next(state_id)=s0 & ! next(state_id)=s0)
```

in the `TRANS` declaration, which will ensure that there is no outgoing edge from the node s in the FSM. (Note that there is no difficulty in representing non-determinism in the action description by this means.)

(The following peculiarity of SMV ought to be noted. Suppose that for a given action description D, the set of labels—the set of all interpretations of $\mathrm{I}(\sigma^a)$ which occur in transitions—is E. It may happen that for a given state s some, but not all members of E are amongst the s-labels. In that case there will be at least one clause of the form

```
  (state_id=s ->   ( action_id=a & next(state_id)=s0))
```

in the `TRANS` declaration. Now, if e^* is amongst the members of E which are not s-labels, then the interpretation of the SMV code requires that the state (s, e^*) have no outgoing edges. In other words, since there has been no reference to e^* in that part of the `TRANS` declaration governing the behaviour of the system from state s, it is assumed, by default, that transitions from (s, e^*) have been ruled out. Yet compare this with the case where there is no clause whatsoever in the `TRANS` declaration which describes the behaviour of our system when it starts in state s. In that case, it is assumed that the system may move into *any* state.—This is the rationale for our inclusion of the clause

```
(state_id=s  ->  next(state_id)=s0 & ! next(state_id)=s0)
```

when the state s cannot make any transitions.)

The final part of the SMV program which we generate from the $\mathcal{C}+$ action description is a translation of the queries. This is precisely the same process as was described for the second implementation.

It is straightforward to program this way of depicting transition systems in SMV, and as with our previous two implementations we have written code which runs alongside CCalc, making use of its predicates to generate the models of Γ_0^D and Γ_1^D which encode the sets of states and, respectively, transitions. As might be expected, the structure of the algorithm which generates the SMV representation of the transition system, in terms of the state variables *state_id* and *action_id*, is very much more perspicuous than the method described for our second implementation in Section 5.3.2.

5.5 Comparison

In this section we will comment on the advantages and disadvantages of the three different approaches we have taken to connecting $\mathcal{C}+$ to model-checking.

The first approach is the least promising. Repeated experiments with many different domains expressed as action descriptions have shown that the translation of a temporal-logic specification in LTL into clausal form, in line with the scheme given for bounded model-checking in Section 2.5.1, tends to produce formulas of great length and complexity. Even when use is made of CCalc's built-in optimization techniques for the translation of well-formed formulas of propositional logic into the conjunctive normal form which SAT-solvers require, the process frequently breaks down, exhausting Prolog's memory. The limitation is severe, as the cardinality of the action description, its signature or the length of the run specified through the transition system do not not need to be large, or the query in LTL very complex, for the translation procedure to cease functioning. Even where the translation of the query succeeds, an original formula in LTL will result in clauses of CNF which each SAT-solver we have used takes prohibitively long to solve.

As a consequence of the way in which $[\![M]\!]_k$ is represented by the clauses CCalc generates for the completion of Γ_1^D, the first implementation does cope with any action description of $\mathcal{C}+$; this is clearly a point in its favour. Currently, however, we are somewhat limited in the language we can use to specify properties for which we wish to model-check; we have only introduced a translation-scheme for LTL, whereas the interfaces to NuSMV which our second and third implementations provide allow us to check for any properties expressible in languages which that model-checker accepts: significantly, in CTL additionally to LTL. Part of this limitation could be lifted, to accommodate ACTL (the universal fragment of CTL); the authors of [PWZ02] adapt bounded model-checking to cope with this temporal logic. Yet even if that adaptation were to be implemented, we would still not have the ability to verify properties expressed in the full branching-time logic of CTL.

The second approach does not require statements of temporal logic to be translated into CNF; as the algorithm produces SMV code which expresses the behaviour of the system we are modelling, we may specify the properties the

system should fulfil in any temporal logic accepted by NuSMV. In order to translate causal laws of $\mathcal{C}+$ action descriptions into SMV code, we have had to impose three restrictions on the form and interactions of those causal laws: the first, to definite action descriptions, must also be made with the first and third approaches, and so cannot count as an advantage of the second. The second restriction has also not, in practice, presented problems, or prevented us from working with any action descriptions. Yet the third is more problematic: whilst action descriptions which do not satisfy it are uncommon, they are not unknown. Another failing of the second method is the difficulty of establishing its correctness. The most we have been able to do towards achieving this is running all three implementations on the same action description, and comparing the results of different queries posed in each; where the query is of the standard form accepted by CCALC's native query language, we also have the results of CCALC as an independent verification. Experiments conducted in this way have always given the same results, which counts as a very good indication that the translation algorithm employed in the second approach is correct. This is, of course, far from being a proof of correctness of the algorithm.

The third approach was prompted by the desire to be able to relax the restrictions on the form of action description which we were forced to impose in the second implementation, whilst retaining the central role of an external model-checker, enabling us to make full use of the expressivity of the two temporal logics LTL and CTL. It also provides an easy method for establishing correctness, since we have a correct method for generating the transition system defined by a $\mathcal{C}+$ action description, and the encoding of that transition system in SMV is direct and transparent. (The generation of the transition system for the $\mathcal{C}+$ action description is, of course, by CCALC.)

Whilst the first implementation soon becomes unusable for large action descriptions or complex queries, it performs well for small domains and short formulas of LTL. The second implementation outputs SMV programs representing finite state machines very quickly, even for large domains, and queries, of course, are produced almost instantaneously: LTL formulas do not need to be translated, and the translation scheme for queries in CCALC's standard input language is very efficient. In the case of the third implementation, the limiting factor for producing SMV code is, of course, the speed with which CCALC and any external SAT-solver it employs can find models of Γ_0^D and Γ_1^D: those models which encode the states and transition of the system. For large, complicated domains, this can take some time, although it is to be borne in mind that this process need only be completed once for each action description, after which any number of queries can be posed and answered using NuSMV.

We conducted experiments using sample domains such as the Zoo World and Farmyard (see Sections 3.9 and 2.1.7 respectively), and other domains whose representations satisfy the restrictions imposed for our second implementation. NuSMV showed no significant difference in the time taken to verify properties on FSMs produced by the second and third implementations. Both of these outperformed the first implementation. The second and third implementations also performed slightly better than CCALC for queries which are expressible in all three cases. In further work we would like to study how our programs cope with much larger domains, in order to attain a clearer view of the advantages and disadvantages of our three implementations.

We have not included the PROLOG code for our three implementations as an

appendix to this thesis, as the file is too large (c. 3800 lines). It is available at `http://www.doc.ic.ac.uk/~rac101/code/ccmc/`.

Chapter 6

Conclusion

This thesis has focused on the action language $\mathcal{C}+$ as a representative of a large family of action languages, logical representations for reasoning about action and change. It has followed two threads of investigation in relation to that action language: that of efficiency, and that of expressivity (in particular, expressivity in relation to temporal distance).

The starting-point of the the first part of the thesis was the intuition that much needless computation was being performed in certain reasoning tasks using the methods of CCalc, or stable model generators, both of which must construct entire models when answering a query about a specific fluent constant at a specific time. It is not always necessary to know everything about the entire history of the system we are modelling. The insight that this comprehensive knowledge is *not* necessary is something which the Event Calculus (when considered with top-down, goal-directed queries) enshrines, and our question was thus: can an Event Calculus style of computation be employed when answering queries of $\mathcal{C}+$ action domains?

In Chapter 3, we presented axioms of a logic program which, when combined with a suitable representation of an action description and specific details about an initial state and narrative of actions through the system, enabled queries to be answered in a goal-directed way, concentrated on causative information relevant to the fluent's value. The main result of the chapter demonstrated the correctness of our approach: stable models of this logic program uniquely define runs through the transition system produced by $\mathcal{C}+$'s normal semantics.

A number of restrictions had to be made on the form of action descriptions, including one prohibiting them from exhibiting what we called a *dependence* amongst fluent constants. Moreover, we have not shown that our goal-directed way of answering queries works with non-deterministic action descriptions However, the class of action descriptions we *can* work with is large and diverse. We illustrated this fact by formalizing a more complex variant of the Yale Shooting Problem, a standard, long-established domain from the literature on reasoning about action and change. We also formalized a variant of the Zoo World, and described how the underlying Prolog of our implementation could be used to improve the representation of domains.

When working with the Event Calculus, there are two stages to reasoning about domains. The first is to check an input domain description, initial state, and narrative of events for consistency, and in the second one proceeds to answer

the queries one wants answered. In $\mathcal{C}+$ when working with CCALC, these two stages are unified: both occur when one finds a model of the underlying, causal-theoretic representation. For us, however, the introduction of computations in the style of the Event Calculus has teased apart these two tasks, and we must go through a separate process of consistency-checking, before the query evaluator of $\mathcal{EC}+$ can be applied. The way in which we execute these consistency checks was described in the text, together with methods for making it more streamlined; these methods were illustrated when the Zoo World was discussed in later sections.

We also conducted a number of experiments on elaborations of the Farmyard domain, in order to compare the performance of our implementation to that of CCALC. We found that, even when we asked our system to produce an entire narrative of events in the way CCALC does of necessity, our system performs better. Finally in Chapter 3, we proved a number of theorems relating $\mathcal{C}+$ and $\mathcal{EC}+$ to a common variant of the Event Calculus expressed as a logic program.

The results we have attained in our work on $\mathcal{EC}+$ have been very encouraging. In the theoretical component of our investigation, we have found what might be characterized as a mid-point (perhaps one of many) between $\mathcal{C}+$ and the Event Calculus: one which embodies the possibilities for styles of computation of the latter, whilst retaining many of the expressive advantages, and the very useful graphical semantics, of the former. More practically, the system we built performs well and will, we hope, be able to be used in many of the contexts previously occupied by the Event Calculus—with all the benefits attendant on the possession of an easily-definable semantics of labelled transition systems.

In Chapter 4, by contrast, we were concerned to make use of the expressivity afforded by $\mathcal{C}+$'s underlying formalism of causal theories, to let us express much more easily, concisely and efficiently, temporally distant interactions between fluents and actions. After a motivating example, we showed how this could be achieved quite straightforwardly by broadening the syntax of causal laws in $\mathcal{C}+$, and shows how the new causal laws should be converted, parameterized by integer-time, into causal rules for solution by a system such as CCALC.

The more involved part of this stage of the work was an investigation into what becomes of the transition systems defined by action descriptions. These graphical correlates of causal laws are at the heart of $\mathcal{C}+$'s utility for us, and we wanted to retain the property that runs through graphical systems defined by action descriptions correspond to models of the relevant causal theory Γ_t^D. Yet this property failed when we first broadened the syntax of laws, and introduced our λ operator. The solution was found by generalizing labelled transition systems to *run systems*, which we defined and then showed, in Theorem 4.4 have the desired property: paths through the run system correspond to models of the causal theory.

We went on to give several illustrations of our constructions, and concluded the chapter by describing how $\mathcal{C}+_{timed}$, our name for the generalization of $\mathcal{C}+$ we introduced, can be married to the deontic concepts of $n\mathcal{C}+$ [SC06], producing coloured run systems.

In Chapter 5 we remained concerned with questions of expressivity, but moved our attention from the causal laws which define a system's behaviour to the query languages we use to answer questions about that behaviour. The query language for $\mathcal{C}+$ which can be used with CCALC is restricted in ways we have found to be inconvenient: it allows reference to the truth of fluents

only at specifically given times, so that it is impossible to ask, for instance, whether something will be true *eventually*, given that such-and-such is true, or this-and-that happens. Yet this sort of proposition is something which is very easily represented in standard temporal logics such as the linear-time LTL, or branching-time CTL. Model-checking is a procedure which takes a graphical representation of a system and verifies whether or not a formula of temporal logic holds in that system, and we wished to exploit this, and the graphical semantics of $\mathcal{C}+$, to apply model-checking methods to this action language. To this end, we proposed, described, and implemented three different methods of model-checking $\mathcal{C}+$ action descriptions. The first supplemented predicates in CCalc by procedures for translating a formula of LTL into conjunctive normal form. The second implementation produced a program of SMV (the input language, defining a finite state machine, for the model-checker NuSMV) which describes the behaviour of the system on a constant-by-constant basis, in an attempt to find close correlates of causal laws in the original action description. The third implementation uses CCalc to construct a complete representation of the labelled transition system defined by the input action description, and then also encodes this transition system, more directly, in SMV. We gave details of all three approaches, and described the limitations, advantages and disadvantages of each.

6.1 Further Work

There are a number of directions for future work, beginning in this thesis, which we would like to pursue.

We are interested in continuing our work on $\mathcal{EC}+$ (Chapter 3) by examining ways in which the restrictions we placed on the form of action descriptions may be relaxed. At the moment, we insist on exogeneity for all action constants of the signature, and no other action dynamic laws may be included in $\mathcal{EC}+$. This is probably too restrictive, and initial attempts at permitting action dynamic laws, in certain circumstances, have proved encouraging. There are also ways in which we ought to be able to allow more kinds of fluent dynamic law; at the moment, the component G of a law F **if** G **after** H can only ever be $\top$, or identical to F (the latter, in the case of laws of inertia). This is certainly too restrictive, and it seems likely that allowing conjunctions to form the components G is possible, as long as a condition similar to that of the absence of *dependence*, which we introduced for static causal laws, is satisfied.

We would also like to investigate the possibility of merging $n\mathcal{C}+$ and $\mathcal{EC}+$, to use an Event Calculus style of computation in reasoning about deontic domains. The difficulty here is likely to be a successful incorporation of the *green-green-green* constraint. We have already had promising results in incorporating the other part of the original $(\mathcal{C}+)^{++}$ language presented in [Ser04]—that involving the representation of *counts as* relations—into our computational framework for $\mathcal{EC}+$. Both of these extensions, for deontic concepts and 'counts-as', are steps towards the greater goal of making action languages such as $\mathcal{C}+$ more suitable for certain sorts of reasoning task involving multi-agent systems.

$\mathcal{EC}+$ can be seen as a marriage of the Event Calculus and $\mathcal{C}+$, and we are also interested in exploring the possibility of other forms of intermediary formalisms, inspired by the execution style of the former, and possessing a graphical seman-

tics similar to the latter. One prominent and useful characteristic of the Event Calculus is the way that the computation of a fluent's value can proceed by jumping back from the current time to a previous time greater than one time-step distant, where an action occurred which may have affected the value of the fluent. In $\mathcal{EC}+$, the analogous process must step back long the time-line, passing through each intervening time-step. We are interested in seeing whether the axioms of $\mathcal{EC}+$ can be adapted to make this stepwise consideration of histories unnecessary. This work is in a very preliminary stage.

In relation to our work on 'distant causation' and $\mathcal{C}+_{timed}$, we feel our work is more self-contained, though there are several unanswered questions we would like to examine. Many of the domains we are enabled to represent more easily in $\mathcal{C}+$ do not involve the full expressivity afforded us by the λ operator; often they make use only of the ability causally to relate actions which are one time-step distant from each other. (This was the case with the 'Reagan and Gorbachev' example we gave in Section 4.3.7.) We would like to see whether the run systems which are generated by this subset of action descriptions of $\mathcal{C}+_{timed}$ can be generated more easily, and to investigate their properties.

Chapter 5 suggests a number of directions for future work. As discussed in Section 5.5, we would like to apply our three implementations to a number of large domains, in order to enable a better appreciation of the different performances of each. We also have ideas for other approaches to connecting action languages and model-checking, which instead of producing SMV code, which NuSMV would then compile down into a propositional formula (for BMC) or a native representation in OBDDs (for symbolic model checking), would move directly to the OBDD representation itself. This work is currently at the planning stage only.

Bibliography

[AEL+04] Varol Akman, Selim T. Erdogan, 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.

[BAPM83] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. In *Advances in Computers*. Academic Press, 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[Bel87] M. Belzer. Legal reasoning in 3-d. In *Proceedings of the first international conference on Artificial intelligence and law*, pages 155–163. ACM Press, 1987.

[BG04] Brandon Bennett and Antony Galton. A unifying semantics for time and events. *Artificial Intelligence*, 153(1-2):13–48, 2004.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Cla78] Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, New York, 1978. Plenum Press.

[CS01] E.M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science, 2001.

[CS05] Robert Craven and Marek J. Sergot. Distant causation in $\mathcal{C}+$. *Studia Logica*, 79(1):73–96, 2005.

[DGKK98] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. Tal: Temporal action logics language specification and tutorial. *Electronic Transactions in Artificial Intelligence*, 2:273–306, 1998.

[DPP04] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004.

[dSA95] Paulo Jorge de Sousa Azevedo. *Techniques for Preventing Recomputation in Logic Programs.* PhD in Computing, Department of Computing, Imperial College London, 1995.

[EL06] Selim T. Erdogan and Vladimir Lifschitz. Actions as special cases. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 377–388. AAAI Press, 2006.

[FL05] Alberto Finzi and Thomas Lukasiewicz. Game-theoretic reasoning about actions in nonmonotonic causal theories. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *LPNMR*, volume 3662 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2005.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[GL93] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.

[GL98] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3, 1998.

[GLL+04] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:49–104, 2004.

[HM87] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[Hog90] Christopher John Hogger. *Essentials of Logic Programming.* Oxford University Press, 1990.

[KM97a] Antonis C. Kakas and Rob Miller. Reasoning about actions, narratives and ramification. *Electron. Trans. Artif. Intell.*, 1:39–72, 1997.

[KM97b] Antonis C. Kakas and Rob Miller. A simple declarative language for describing narratives with actions. *The Journal of Logic Programming*, 31(1-3):157–200, 1997.

[KMT01] Antonis C. Kakas, Rob Miller, and Francesca Toni. $\mathcal{E}$-RES: Reasoning about actions, events and observations. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 2001.

[KS86] R.A. Kowalski and M.J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[Lif94] Lifschitz, V. Circumscription. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3*, volume 3, pages 297–352. Oxford University Press, 1994.

[Llo87] John Lloyd. *Foundations of Logic Programming*. Springer, 2 edition, 1987.

[LR06] Vladimir Lifschitz and Wanwan Ren. A modular action description language. In *AAAI*. AAAI Press, 2006.

[McC80] McCarthy, J. Circumscription - A form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[MH69] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[MS02] Rob Miller and Murray Shanahan. Some alternative formulations of the event calculus. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 452–490. Springer-Verlag, 2002.

[MT97] Norman McCain and Hudson Turner. Causal theories of action and change. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 460–465, Menlo Park, California, 1997. AAAI Press.

[Mue06a] Erik T. Mueller. *Commonsense Reasoning*. Morgan Kaufmann, 2006.

[Mue06b] Erik T. Mueller. Event calculus and temporal action logics compared. *Artificial Intelligence*, 170(11):1017–1029, 2006.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theor. Comput. Sci.*, 13:45–60, 1981.

[PWZ02] Wojciech Penczek, Bozena Wozna, and Andrzej Zbrzezny. Bounded Model Checking for the Universal Fragment of CTL. *Fundamenta Informaticae*, 51(1-2):135–156, 2002.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[SC05a] Marek Sergot and Robert Craven. Logical Properties of Nonmonotonic Causal Theories and the Action Language $\mathcal{C}+$. Technical Report 2005/5, Department of Computing, Imperial College London, 2005.

[SC05b] Marek Sergot and Robert Craven. Some logical properties of nonmonotonic causal theories. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *Lecture Notes in Computer Science*, pages 198–210. Springer, 2005.

[SC06] Marek Sergot and Robert Craven. The deontic component of action language $n\mathcal{C}+$. In Lou Goble and John-Jules Ch. Meyer, editors, *DEON*, volume 4048 of *Lecture Notes in Computer Science*, pages 222–237. Springer, 2006.

[Ser04] Marek Sergot. $(\mathcal{C}/\mathcal{C}+)^{++}$: An action language for modelling norms and institutions. Technical Report 2004/8, Department of Computing, Imperial College London, 2004.

[Sha90] Murray Shanahan. Representing continuous change in the event calculus. In *ECAI*, pages 598–603, 1990.

[Sha95] Murray Shanahan. A Circumscriptive Calculus of Events. *Artificial Intelligence*, 77:249–284, 1995.

[Sha97] Murray Shanahan. *Solving the Frame Problem.* The MIT Press, 1997.

[Sha99] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today*, Lecture Notes in Computer Science, pages 409–430. Springer, 1999.

[Sha00] Murray Shanahan. An Abductive Event Calculus Planner. *The Journal of Logic Programming*, 44:207–239, 2000.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

Appendix A

The Farmyard Resurrection domain

Here is the PROLOG source file used for the $\mathcal{EC}+$ representation of our 'Farmyard Resurrection' domain. It is possible to make the specification of the signature (in particular, the definition of `domain/2`) much more concise, but we have chosen to define the predicates at greater length for reasons of clarity.

```
% ------------ fluent constants

flu_constant(alive(C)) :-
  character(C).
flu_constant(loaded).
flu_constant(loc(C)) :-
  character(C).
flu_constant(smiling(C)) :-
  character(C).
flu_constant(target).

% ------------ action constants

act_constant(aim).
act_constant(miracle(C)) :-
  character(C).
act_constant(load).
act_constant(shoot).
act_constant(walk(C)) :-
  character(C).

% ------------ domains for fluent constants

domain(alive(_), V) :-
  boolean(V).
domain(loaded, V) :-
  boolean(V).
```

```
domain(loc(_), V) :-
  location(V).
domain(smiling(C)) :-
  boolean(V).
domain(target, V) :-
  (location(V) ; V = none).

% ------------ domains for action constants

domain(aim, V) :-
  (location(V) ; V = ff).
domain(miracle(_)) :-
  boolean(V).
domain(load, V) :-
  boolean(V).
domain(shoot, V) :-
  boolean(V).
domain(walk(_), V) :-
  (location(V) ; V = ff).

% ------------ parameters for the signature

character(bill).
character(turkey).

location(barn).
location(house).
location(field).

boolean(tt).
boolean(ff).

% ------------ inertia

all_inertial.

% ------------ static causal laws

causes(alive(C)=tt,   [smiling(C)=tt]).
causes(smiling(C)=ff, [alive(C)=ff]).

% ------------ fluent dynamic causal laws

causes(alive(X)=ff,   [shoot=tt],      [loaded=tt,
                                        target=L,
                                        loc(X)=L]).
causes(loaded=ff,     [shoot=tt],      [loaded=tt]).
causes(loaded=tt,     [load=tt],       []).
causes(loc(C)=L,      [walk(C)=L],     []) :- location(L).
causes(smiling(X)=tt, [miracle(X)=tt], [alive(X)=ff]).
```

```
causes(target=L,      [aim=L],         []) :- location(L).
causes(target=none,   [load=tt],       []).

% ------------ nonexecutability laws

nonexecutable([walk(C)=L], [loc(C)=L]) :- location(L).
nonexecutable([walk(C)=L], [alive(C)=ff]) :- location(L).

% ------------ initial state

init(alive(C)=tt) :- character(C).
init(loaded=ff).
init(loc(bill)=house).
init(loc(turkey)=barn).
init(smiling(bill)=ff).
init(smiling(turkey)=tt).
init(target=none).

% ------------ narrative of events

happens(load=tt,             0).
happens(aim=field,           1).
happens(walk(bill)=field,    2).
happens(shoot=tt,            3).
happens(walk(turkey)=house,  4).
happens(load=tt,             6).
happens(walk(turkey)=field,  6).
happens(walk(turkey)=barn,   7).
happens(aim=barn,            7).
happens(miracle(bill)=tt,    8).
happens(shoot=tt,            9).
happens(walk(bill)=house,   10).
happens(miracle(turkey)=tt, 11).
```

Appendix B

The Zoo World

This is the action description for our formalization of the Zoo World, as presented in Section 3.9 of the thesis. We here present the $\mathcal{EC}+$ representation as PROLOG source code.

B.1 Action Description

The source file is divided into three main sections. The first section represents more general knowledge about the Zoo World, and the second section encodes the details of specific examples. Thus, the first section includes causal laws about what happens when animals move; the second section tells what the details of a specific zoo are: the locations, animals, topology, and so on; and the third section sets out the initial state and narrative of events. The action description which forms the first section is parametric on the information about zoo specifics which is contained in the second.

The first section is by far the largest. First, there is a specification of the signature of Zoo World action descriptions.

```
% ------------ fluent constants

flu_constant(opened(G)) :-
  gate(G).
flu_constant(accessible(P1,P2)) :-
  position(P1),
  position(P2),
  neighbour(P1,P2).
flu_constant(pos(A)) :-
  animal(A,_).
flu_constant(mounted(H)) :-
  animal(H,human).

% ------------ action constants

act_constant(move(A)) :-
  animal(A,_).
```

```
act_constant(open(H,G)) :-
  animal(H, human),
  gate(G).
act_constant(close(H,G)) :-
  animal(H, human),
  gate(G).
act_constant(mount(H)) :-
  animal(H, human).
act_constant(get_off(H,A)) :-
  animal(H,human),
  animal(A,_),
  H \= A.

% ------------ domains for fluent constants

domain(opened(_), V) :-
  boolean(V).
domain(accessible(_,_), V) :-
  boolean(V).
domain(pos(_),P) :-
  position(P).
domain(mounted(_),V) :-
  (animal(V,_) ; V = ff).

% ------------ domains for action constants

domain(move(_),V) :-
  (position(V) ; V = ff).
domain(open(_,_),V) :-
  boolean(V).
domain(close(_,_),V) :-
  boolean(V).
domain(mount(H),V) :-
  ((animal(V,_),V\=H) ; V = ff).
domain(get_off(_,_),V) :-
  (position(V) ; V = ff).
```

The definitions of constants and their domains above depend both on some more general knowledge (such as that, in the words of the original specification, "[a]dult members of large species are large animals"), and also on more specific knowledge about what animals there are in a given zoo. The general information is given below.

```
% ------------ Boolean truth-values

boolean(tt).
boolean(ff).

% ------------ Other Predicates
```

```
position(P) :-
  loc(P,_).

neighbour(P1, P2) :-
  neighbour(P1,P2,l).

neighbour(P1,P2,_) :-
  sides(_,P1,P2).

neighbour(P1,P2,l) :-
  neighbour(P2,P1,r).

neighbour(P1,P2,_) :-
  nb(P1,P2).

adult(A) :-
  animal(A,_),
  \+ infant(A).

large_animal(A) :-
  adult(A),
  animal(A,S),
  large_species(S).

small_animal(A) :-
  animal(A,_),
  \+ large_animal(A).
```

There now follow the causal laws of the action description: first a record of which fluents are inertial, then static laws whose heads are not $\perp$, then fluent dynamic laws whose heads are not $\perp$. (Recall that all action constants are automatically deemed to be exogenous in our action descriptions; this does not need to be represented explicitly.)

```
% ------------ Inertia

inertial(opened(_)=V) :-
  boolean(V).
inertial(pos(_)=P) :-
  position(P).
inertial(mounted(_)=_).

% ------------ Static Causal Laws (constants in head)

causes(accessible(P1,P2)=tt, []) :-
  neighbour(P1,P2),
  \+ sides(_,P1,P2),
  \+ sides(_,P2,P1).
```

```
causes(accessible(P1,P2)=tt, [opened(G)=tt]) :-
  (sides(G,P1,P2) ; sides(G,P2,P1)).
causes(accessible(P1,P2)=ff, [opened(G)=ff]) :-
  (sides(G,P1,P2) ; sides(G,P2,P1)).
causes(pos(H)=P, [mounted(H)=A,pos(A)=P]) :-
  animal(A,_).

% ------------ Dynamic Causal Laws (constants in head)

causes(pos(A)=P, [move(A)=P], []) :-
  position(P).
causes(opened(G)=tt, [open(_,G)=tt], []).
causes(opened(G)=ff, [close(_,G)=tt], []).
causes(pos(H)=P, [mount(H)=A,move(A)=P1], [pos(A)=P]) :-
  position(P1).
causes(pos(H)=P, [get_off(H,A)=P,move(A)=ff], []) :-
  position(P).
causes(mounted(H)=ff, [get_off(H,A)=P,move(A)=ff], []) :-
  position(P).
causes(mounted(H)=A, [mount(H)=A,move(A)=ff], []) :-
  animal(A,_).
```

Now, the causal laws which have $\bot$ as their head, and which are used when we check that the action description, initial state and narrative of events define a run through the transition system the former alone determines. The knowledge which gave rise to each constraint has been printed above it, as a comment; the breakdown into individual statements is owed to [AEL+04]. We first give the static, and then the fluent dynamic causal laws.

```
% ------------ Never Laws

% Two large animals can not occupy the same position,
% except if one of them rides on the other

% (case 1: neither are human)

never([pos(A1)=P,pos(A2)=P]) :-
  large_animal(A1),
  large_animal(A2),
  A1 @< A2,
  animal(A1,S1),
  animal(A2,S2),
  S1 \= human,
  S2 \= human.

% (case 2: one is human)

never([pos(A1)=P,pos(A2)=P,mounted(A)=V]) :-
  large_animal(A1),
```

```
  large_animal(A2),
  A1 @< A2,
  (animal(A1,human),
   animal(A2,S),
   S \= human,
   A = A1,
   domain(mounted(A),V),
   V \= A2
   ;
   animal(A2,human),
   animal(A1,S),
   S \= human,
   A = A2,
   domain(mounted(A),V),
   V \= A1
  ).

% (case 3: both are human)

never([pos(A1)=P,pos(A2)=P,mounted(A1)=V1,mounted(A2)=V2]) :-
  large_animal(A1),
  large_animal(A2),
  A1 @< A2,
  animal(A1,human),
  animal(A2,human),
  domain(mounted(A1),V1),
  domain(mounted(A2),V2),
  V1 \= A2,
  V2 \= A1.

% a human can only be mounted on a large animal

never([mounted(_)=A]) :-
  small_animal(A).

% a large human cannot be mounted on a human
% (this we could do in the signature)

never([mounted(H1)=H2]) :-
  large_animal(H1),
  animal(H2,human),
  animal(H1,human).

% an animal can be mounted by at most one human at a time

never([mounted(H1)=A,mounted(H2)=A]) :-
  animal(H1,human),
  animal(H2,human),
  H1 \= H2,
  animal(A,_).
```

```
% a human cannot be mounted on a human who is mounted
% (should we insist H1, H2 are humans?)

never([mounted(H1)=H2,mounted(H2)=H3]) :-
  animal(H3,_).

% ------------ Nonexecutable Laws

% In one unit of time, an animal can move to one of the positions
% accessible from its present one, or stay in the position
% here it is.  Moves to non-accessible positions are
% never possible
% (This can be constrained by the following.)

nonexecutable([move(A)=P],[pos(A)=P1,accessible(P,P1)=ff]) :-
  position(P).
nonexecutable([mount(H)=A],[pos(A)=P1,pos(H)=P2,
                            accessible(P1,P2)=ff]) :-
  animal(A,_).
nonexecutable([get_off(H,A)=P1],[pos(A)=P2,
                                 inaccessible(P1,P2)=ff]).

% A concurrent move where animal A moves into a position at the
% same time as animal B moves out of it, is only possible if
% at least one of A and B is a small animal.
% Exceptions for (failed) mount actions.

nonexecutable([move(A1)=P1,move(A2)=P2],[pos(A2)=P1]) :-
  large_animal(A1),
  large_animal(A2),
  A1 \= A2,
  neighbour(P1,P2).

% Two large animals cannot pass through a gate at the same time
% (neither in the same direction nor opposite directions)

nonexecutable([move(A1)=P1,move(A2)=P2],
                           [pos(A1)=P2,pos(A2)=P1]) :-
  large_animal(A1),
  large_animal(A2),
  A1 @< A2,
  (sides(_,P1,P2) ; sides(_,P2,P1)).

nonexecutable([move(A1)=P,move(A2)=P],[pos(A1)=P1,pos(A2)=P1]) :-
  large_animal(A1),
  large_animal(A2),
  A1 @< A2,
  (sides(_,P,P1) ; sides(_,P1,P)).
```

```
% While a gate is closing, an animal cannot pass through it

nonexecutable([move(A)=P1,close(_,G)=tt],[pos(A)=P2]) :-
  (sides(G,P1,P2) ; sides(G,P2,P1)).

% an animal can't move to the position where it is now

nonexecutable([move(A)=P],[pos(A)=P]).

% a human riding an animal cannot perform the move action

nonexecutable([move(H)=P],[mounted(H)=A]) :-
  animal(A,_),
  position(P).

% a human cannot open a gate if he is not located at
% a position to the side of the gate

nonexecutable([open(H,G)=tt],[pos(H)=P]) :-
  position(P),
  \+ sides(G,P,_),
  \+ sides(G,_,P).

% a human cannot open a gate if he is mounted on an animal

nonexecutable([open(H,_)=tt],[mounted(H)=A]) :-
  animal(A,_).

% a human cannot open a gate if it is already opened

nonexecutable([open(H,G)=tt],[opened(G)=tt]).

% a human cannot close a gate if he is not located at
% a position to the side of the gate

nonexecutable([close(H,G)=tt],[pos(H)=P]) :-
  position(P),
  \+ sides(G,P,_),
  \+ sides(G,_,P).

% a human cannot close a gate if he is mounted on an animal

nonexecutable([close(H,_)=tt],[mounted(H)=A]) :-
  animal(A,_).

% a human cannot close a gate if it is already closed

nonexecutable([close(_,G)=tt],[opened(G)=ff]).

% a human already mounted cannot attempt to mount
```

```
nonexecutable([mount(H)=A], [mounted(H)=A1]) :-
  animal(A,_),
  animal(A1,_).

% a human cannot attempt to mount an animal already mounted
% by a human
% (question of whether we should insist that H is a human)

nonexecutable([mount(H)=A], [mounted(H1)=A]) :-
  animal(A,_).

% a human cannot attempt to mount an animal if
% the human is already mounted by a human
% (question of whether we should insist that H is a human)

nonexecutable([mount(H)=A], [mounted(H1)=H]) :-
  animal(A,_).

% A human cannot attempt to mount a human who is mounted
% (this is covered by an earlier case)

% get_off cannot be performed by a human not riding an animal

nonexecutable([get_off(H,A)=P],[mounted(H)=X]) :-
  animal(A,_),
  position(P),
  domain(mounted(H),X),
  X \= A.

% a human cannot attempt to get_off to an inaccessible position

nonexecutable([get_off(_,A)=P],[pos(A)=P1,accessible(P,P1)=ff]).
```

We have included the details of a specific Zoo World domain, for illustrative purposes. This is the zoo which was described in Section 3.9 of this thesis, and whose transition system was depicted in Figure 3.8.

```
% ------------ Topology and Population
%
% (here we include facts:
%    gate/1
%    cage/1
%    nb/2
%    sides/3
%    loc/2
%    animal/2
%    large_species/1
%    infant/1)
```

```
gate(gate).

cage(cage).

sides(gate,1,2).

loc(1,cage).
loc(2,outside).

animal(ahab,human).
animal(moby,whale).

large_species(whale).
large_species(human).
```

Finally, the initial state and narrative of a (very short) sample run through the system.

```
% ------------ Initial State

init(opened(gate)=ff).
init(accessible(P1,P2)=ff)
  :- neighbour(P1,P2).
init(pos(moby)=2).
init(pos(ahab)=1).
init(mounted(ahab)=ff).

% ------------ Narrative

happens(open(ahab,gate)=tt,   1).
happens(mount(ahab)=moby,     2).
happens(move(moby)=1,         3).
happens(get_off(ahab,moby)=2, 4).
happens(close(ahab,gate)=tt,  5).
```

B.2 Domain Constraints

```
/*
 *
 * constraint(_) is true when something goes wrong.
 *
 */

% each position is included in precisely one location
% (we can ensure this by construction)
```

```
constraint(0) :-
  position(P),
  loc(P,L1),
  loc(P,L2),
  L1 \= L2.

% each position must have at least one neighbour

constraint(1) :-
  position(P),
  \+ neighbour(P,_).

% the neighbour relation is irreflexive

constraint(2) :-
  neighbour(P,P).

% the neighbour relation is symmetric

constraint(3) :-
  neighbour(P1,P2),
  \+ neighbour(P2,P1).

% one location is the outside

constraint(4) :-
  \+ loc(_,outside).

% all other locations are cages

constraint(5) :-
  loc(_,L),
  L \= outside,
  \+ cage(L).

% two positions are the sides of a gate

constraint(6) :-
  sides(_,P1,P2),
  ( (\+ position(P1)) ; (\+ position(P2)) ).

% the positions which form the sides of a gate
% must occupy different locations

constraint(7) :-
  sides(_,P1,P2),
  loc(P1,L),
  loc(P2,L).
```

```
% no two gates have the same sides

constraint(8) :-
  sides(G1,P11,P12),
  sides(G2,P21,P22),
  G1 @< G2,
  (P11 = P21, P12 = P22 ; P11 = P22, P12 = P21).

% two positions are neighbours if they are the sides of a gate

constraint(9) :-
  sides(_,P1,P2),
  \+ neighbour(P1,P2).

% two positions in different locations are neighbours
% only if they are the sides of a gate

constraint(10) :-
  loc(P1,L1),
  loc(P2,L2),
  P1 @< P2,
  L1 \= L2,
  \+ sides(_,P1,P2),
  \+ sides(_,P2,P1),
  neighbour(P1,P2).

% one of the species is human

constraint(11) :-
  \+ animal(_,human).

% each animal belongs to exactly one species

constraint(12) :-
  animal(A,S1),
  animal(A,S2),
  S1 \= S2.

% some species are large, some are not

constraint(13) :-
  ( \+ (animal(_,S),large_species(S))
    ;
    \+ (animal(_,S),\+ large_species(S))).

% adult members of large species are large animals

constraint(14) :-
  large_species(S),
  animal(A,S),
```

```
  adult(A),
  \+ large_animal(A).

% there is at least one human in each scenario

constraint(15) :-
  \+ animal(_,human).

% every animal can only do one action at a time

done_by(move(A), A).
done_by(open(H,_), H).
done_by(close(H,_),H).
done_by(mount(H),H).
done_by(get_off(A,_),A).

constraint(16) :-
  happens(C1,V1,T),
  V1 \= ff,
  V1 \= none,
  happens(C2,V2,T),
  V2 \= ff,
  V2 \= none,
  a(C1,V1) \= a(C2,V2),
  done_by(C1,A),
  done_by(C2,A).

% a human cannot attempt to mount a small animal

constraint(17) :-
  happens(mount(_),A,_),
  small_animal(A).

% a large human cannot attempt to mount a human

constraint(18) :-
  happens(mount(HL),H,_),
  animal(H,human),
  large_animal(HL).
```