KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A - 3001 Leuven (Heverlee)

OPEN LOGIC PROGRAMMING AS A KNOWLEDGE REPRESENTATION LANGUAGE FOR DYNAMIC PROBLEM DOMAINS

Jury :
Prof. Dr. ir. R. Govaerts, voorzitter
Prof. Dr. D. De Schreye, promotor
Prof. Dr. ir. M. Bruynooghe
Prof. Dr. B. Demoen
Prof. Y. Deville (UCL)
Dr. M. Shanahan (Queen Mary College, London)
Dr. D. Theseider Dupré (Univ. of Torino)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen
door

Kristof VAN BELLEGHEM

U.D.C. 681.3*I23

December 1997

©Katholieke Universiteit Leuven - Faculteit Toegepaste Wetenschappen
Arenbergkasteel, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemmimg van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/1997/7515/27

ISBN 90-5682-093-1

Open Logic Programming as a Knowledge Representation Language for Dynamic Problem Domains

Kristof Van Belleghem

Department of Computer Science, K.U. Leuven

Abstract

Open logic programming (OLP) is a recently developed knowledge representation language combining logic programming and first order logic, and inheriting the advantages of both formalisms. It provides considerable expressive power useful for representing a large class of problem domains in a declarative, problem-independent way. A resolution-based procedure allows for solving a wide range of problems given such a general domain representation.

In a first part of this thesis we show the general applicability of OLP by comparing it with a class of widely used knowledge representation languages, the description logics, and showing that OLP is a generalisation of these languages incorporating the same principles.

Further we apply OLP to domains that change over time and in which the evolution of the domain is essential. On one hand, we show how the formalism can deal with open fundamental problems in this research area, in particular the different aspects of the frame problem. We analyse the constructs needed for a good representation of dynamic domains, and develop a high-level language based on OLP and incorporating these constructs. We also compare OLP formalisations of the two most widely used formalisms in temporal reasoning, Situation Calculus and Event Calculus. We study the consequences of the differences between the two formalisms for knowledge representation and create a new formalism which generalises both of them and inherits their advantages.

On the other hand, we use the OLP Event Calculus outside of its usual area of application. We extend the formalism for dealing with qualitative continuous change using the OLP machinery for dealing with incomplete information. We use OLP Event Calculus to represent a temporal knowledge base containing incomplete data and show how the functionality of such a knowledge base can be provided using the existing procedure for OLP. Finally, we use OLP Event Calculus for protocol specification and compare it with existing specialised languages for this task. These three applications illustrate how the formalism can be easily extended for use in a wide class of less traditional problem domains.

Acknowledgements

Now that I have completed this Ph.D. thesis, I would like to thank all the people who directly or indirectly have helped me to reach this stage.

In the first place I thank my promoter, Professor Danny De Schreye, for encouraging me to start this research and even more for continually pointing out ways to improve the contents and presentation of candidate publications. Many thanks also go to Marc Denecker for four years of considerable cooperation. He taught me to be mathematically precise, never allowing me to ignore any seemingly insignificant detail. Though I almost never agreed with him at first, in the end I often had to admit the soundness of his arguments.

I thank the members of my jury, Professors Maurice Bruynooghe, Bart Demoen, Yves Deville, Murray Shanahan and Daniele Theseider Dupré, for reading my thesis and providing me with many detailed and valuable comments. Daniele Theseider Dupré I also thank for the cooperation on the work on $\mathcal{ER}$. He would not allow us to be satisfied with a solution that just looked fine from our perspective, but always studied any idea from a number of different points of view.

Frank Piessens and Guy Duchatelet deserve my thanks for their considerable contributions to the work on protocol specification. Frank has moreover had the courage to partially check the protocol's correctness proof.

For interesting discussions concerning various issues in temporal reasoning I thank many people, but especially Bern Martens, Murray Shanahan, Rob Miller, Tony Kakas, Robert Kowalski and Vladimir Lifschitz. These discussions have helped me gain more insight in other points of view on the problems tackled here, and I hope they have resulted in important improvements in my own approach.

Many thanks go to my office-mates, Michael Leuschel and Sofie Verbaeten, not only for the many discussions on logic and logic programming, but even more for creating an agreeable working atmosphere, so that there was always some light shining even in those darkest days when the logic would not cooperate.

In this respect I of course also thank Wouter Camps and Gorik De Samblanx, as well as the people of our role-playing group. They offered many much welcomed alternatives to logic and computer science, and made sure I never lost sight of the rest of the world. I now wish Gorik luck in the final stage of his own Ph.D.

Last but not least, thanks go to my parents, thanks to whom I could ignore many of the everyday problems when this thesis required all of my attention. They encouraged me from beginning to end, and made sure I could work on this thesis in ideal circumstances.

Finally, I would like to thank the IWT (Vlaams instituut voor de bevordering van het wetenschappelijk-technologisch onderzoek in de industrie) for their support for this thesis, which is of course much appreciated.

Truth knows no special time. Its hour is now and always.

Albert Schweitzer

Contents

1 Introduction 1

1.1 Knowledge Representation and Open Logic Programming . . . 1

1.2 Overview of the Thesis . . . 3

2 Technical Background 7

2.1 First Order Logic . . . 7

2.2 Logic Programming . . . 10

2.3 Open Logic Programming . . . 12

2.4 Procedural Aspects . . . 19

3 A Formal Correspondence between Open Logic Programming and Description Logics 29

3.1 Introduction . . . 29

3.2 The Ideas behind Description Logics . . . 30

3.3 Syntax and Semantics of Description Logics . . . 31

3.4 Mapping Description Logic Theories to OLP . . . 33

3.4.1 Mapping Function . . . 33

3.4.2 Description Logics as Sublanguages of OLP . . . 39

3.5 Comparison of SLDNFA to a DL Algorithm . . . 41

3.6 Discussion . . . 47

4 Time in Knowledge Representation 49

4.1 Outline of the Problem . . . 49

4.2 The Event Calculus . . . 52

4.2.1 Formalisation . . . 52

4.2.2 Reasoning on Event Calculus Specifications . . . 57

4.2.3 Support for Event Calculus in SLDNFA . . . 59

5 **Comparing and Integrating Event and Situation Calculus** 61
5.1 Introduction and Motivation . . . 61
5.2 Formalisation of the Calculi . . . 63
5.2.1 The Situation Calculus . . . 63
5.2.2 The Event Calculus . . . 65
5.2.3 An Example Problem Domain . . . 67
5.3 Counterfactual Reasoning in Situation Calculus and Event Calculus . . . 68
5.4 A Generalised Calculus . . . 71
5.4.1 The New Calculus . . . 71
5.4.2 Application . . . 73
5.4.3 Introducing Situations . . . 74
5.4.4 The Application Revisited . . . 79
5.5 Relation to the Original Calculi . . . 80
5.6 Restrictions of Situation Calculus in Counterfactual Reasoning . . . 87
5.7 Discussion . . . 92

6 **Knowledge Representation in OLP Event Calculus** 97
6.1 Representing Continuous Change . . . 97
6.1.1 Introduction . . . 97
6.1.2 Elementary Changes . . . 99
6.1.3 Some Applications . . . 104
6.1.4 Changes Caused by Multiple Influences . . . 106
6.1.5 Discussion . . . 110
6.2 A Framework for Temporal Knowledge Bases . . . 112
6.2.1 Introduction . . . 112
6.2.2 Specification of the Knowledge Base . . . 113
6.2.3 Representing the Data in OLP Event Calculus . . . 115
6.2.4 Basic Functionality . . . 118
6.2.5 A Detailed Example . . . 122
6.2.6 Advanced Functionality . . . 124
6.2.7 Discussion . . . 127
6.3 Protocol Specification . . . 129
6.3.1 Introduction . . . 129
6.3.2 Preliminaries and Notation . . . 130
6.3.3 Specification of a Sliding Window Protocol . . . 131
6.3.4 Translating Process Algebra Specifications to Event Calculus . . . 141
6.3.5 Discussion . . . 142

7 A High-Level Language for Representing Dynamic Domains 147

7.1 Introduction 147
7.2 The Syntax of $\mathcal{ER}$ 150
7.3 The Semantics of $\mathcal{ER}$ 157
7.3.1 Principle of Inductive Definition 160
7.3.2 Tackling the Ramification Problem 161
7.3.3 Formal Semantics of $\mathcal{ER}$ 167
7.3.4 Properties of $\mathcal{ER}$ 169
7.4 Examples of Various $\mathcal{ER}$-contributions 171
7.4.1 Simultaneous Actions 171
7.4.2 Incomplete Narrative Information 172
7.5 Mapping $\mathcal{ER}$ to OLP Event Calculus 174
7.5.1 Mapping Predicates 174
7.5.2 Dealing with the Domain-Independent Conditions 175
7.5.3 Mapping Π_p to FOL Axioms 177
7.5.4 Mapping Π_e to Program Clauses 177
7.5.5 A Detailed Example and some Remarks 178
7.5.6 Equivalence Proof 179
7.6 Dealing with Nondeterminism in $\mathcal{ER}$ 182
7.6.1 Mapping Nondeterministic Effect Rules to OLP 187
7.7 $\mathcal{ER}$ Compared with the Approach of Thielscher 192
7.8 Influence Information 199
7.8.1 Generating Effect Propagation Rules 200
7.8.2 A More Uniform Notation 207
7.8.3 Comparing our Method with Thielscher's 211
7.9 Delayed Causation 216
7.9.1 Mapping Delayed Effect Rules to OLP 222
7.10 Related work 225
7.11 Conclusion 228

8 Conclusion 231

A The Lloyd-Topor Transformation 245

B Proof of Lemma 5.5.3 249

C Correctness Proof of the Sliding Window Protocol Specification 255

C.1 An Invariant Relation . . . 256

C.1.1 Specification . . . 256

C.1.2 Proof . . . 257

C.2 Proof of the Protocol's Correctness . . . 263

C.3 Deadlock-freeness of the Protocol . . . 265

S Samenvatting i

S.1 Inleiding . . . i

S.2 Verband van OLP met Terminologische Talen . . . iv

S.3 De Rol van Tijd in Kennisrepresentatie . . . vi

S.4 Een Integratie van Event en Situation Calculus . . . ix

S.5 Kennisrepresentatie in OLP Event Calculus . . . xi

S.5.1 Voorstellen van Continue Verandering . . . xi

S.5.2 Een Algemene Voorstelling voor Temporele Kennisbanken . . . xiii

S.5.3 OLP Event Calculus als Protocolspecificatietaal . . . xv

S.6 Een Hoog-niveau Representatietaal voor Dynamische Probleemdomeinen . . . xvii

S.7 Besluit . . . xxii

Chapter 1

Introduction

1.1 Knowledge Representation and Open Logic Programming

Over the years the growth in complexity and required flexibility of software has given rise to new programming paradigms. To keep the complexity under control, new programming languages are closer to the human programmer and further from the computer hardware. To offer the required flexibility, programming is seen to move from describing a solution to a specific problem to describing the relevant parts of the problem domain. The object-oriented paradigm as well as declarative languages like functional and logic programming are important steps in this direction.

This trend is present in the entire software domain. However, the need for flexibility is taken to the far end in the field of artificial intelligence, where computers need to function as more and more autonomous agents operating in changing environments without human interaction. For applications in this field the role of programming turns into one of representing the available knowledge, which the agent should use in various — intelligent — ways. To achieve this, two issues are of extreme importance: on the one hand, we need declarative languages able to represent knowledge in a correct, natural and concise way. On the other hand, these languages must incorporate flexible general reasoning procedures allowing agents to perform a wide range of tasks. This thesis is essentially only concerned with the issue of correct knowledge representation. We do indicate where appropriate how reasoning tasks on the generated representations can be soundly performed, but the algorithms are only of theoretical importance. We do not aim at efficient implementations in this thesis.

Classical logic is probably the best-known formal knowledge representation language. It is universally applicable and has a precise, natural semantics. This declarative nature of logic makes it possible to check rather easily whether particular parts of a logical representation are correct or not. Different parts of the representation can be checked entirely independent of each other. The development of automated theorem proving and reasoning techniques on logic theories has given rise to logic programming languages, in which problem domains can be declaratively represented and problems quite efficiently solved by performing deduction on the theory.

As a knowledge representation language, logic programming is both stronger and weaker than classical first order logic. It is stronger due to an implicit *closed world assumption*: intuitively, in logic programming it is automatically assumed that anything which is not mentioned, is not true. This is a very common assumption in natural language: as an example, if it is given that on a particular table there are three blocks A, B and C, and that A is on B, then one tends to assume that there are no other blocks and that C is not on or under A or B. The closed world assumption allows one to specify only what is true, without the need of adding explicitly everything that is not true (which in some cases, in particular when the domain is infinite, is not even possible). As a result, logic programming can represent concepts that cannot be represented in first order logic. The same closed world assumption, however, makes logic programming unsuitable for representing incomplete information: given a closed world assumption, everything is uniquely determined. In the above example, one would not be able to represent that C's position relative to A and B is unknown: if it is not explicitly specified that C is on A or on B, then it is assumed not to be on either block.

As knowledge on any problem domain is very rarely complete, the strength of logic programming is also its most considerable weakness, in particular since incompletely specified domains are by far the most interesting. Given a complete specification, everything is fixed anyway, so the only reasoning task to be performed on such a theory is deduction, i.e. checking whether some statement is true or not. Given a theory with unspecified parts, other reasoning paradigms are of much interest, in particular abduction and model generation. Abduction for example generates assumptions on the incompletely specified part of the theory to explain certain observations. Model generation is a particular application of abductive reasoning, in which a model for the entire set of data is generated.

Open logic programming (OLP) combines the advantages of logic programming with those of classical first order logic by essentially integrating the two formalisms. This makes OLP suitable for representing a very wide range of problem domains in which parts can be completely specified and

other parts incompletely. Moreover reasoning procedures on open logic programs allow for performing many different tasks, both deductive and abductive, on a given representation, thus avoiding the need to encode the same information in many different ways for different tasks.

1.2 Overview of the Thesis

In this thesis we study the expressive power of open logic programming as a general knowledge representation language, and in particular for representing knowledge in the important domain of *temporal reasoning*. This research area is concerned with representing domains that change over time. The major issue to be dealt with in this context is correctly relating the states of a problem domain at different time points. The temporal reasoning research community struggles with many open problems, to which we hope to provide useful answers using open logic programming. Note that the issues in temporal reasoning are relevant in any application where an agent interacts with some environment: any real problem domain is subject to change, if not by itself then at the very least by the actions of the agent performing its designated tasks. The agent must know how its actions will influence the world around it to be able to do anything useful. Hence, any agent requires a notion of time and change in some form or other.

In Chapter 2 we provide the necessary technical background on first order logic, logic programming and open logic programming as declarative languages and on the procedure used for reasoning on open logic programming. In Chapter 3 we show the expressive power of OLP and its suitability for general knowledge representation by formally comparing it with Description Logics, a class of languages specifically designed for representing knowledge in expert systems. We show that description logics correspond very closely to particular subsets of OLP both in general structure and at the level of language constructs. Moreover we show that the procedures used for reasoning on description logics can be seen as specialised instances of the general OLP procedure.

From Chapter 4 on we focus on knowledge representation in temporal reasoning. We start by giving a brief introduction to temporal reasoning in artificial intelligence and we introduce the Event Calculus formalism which we will use in the context of OLP for representing temporal domains. In Chapter 5 we extensively justify this choice and indicate some of its implications by comparing the Event Calculus in detail with the most widely used logic-based formalism in this domain, the Situation Calculus. We show in which contexts the differences between the formalisms are important, and build a new formalism which generalises them both.

In Chapter 6 we study the use of OLP Event Calculus outside of the classical setting for which it was intended, as an indication of the general applicability of this framework. First we extend the Event Calculus for qualitative reasoning on continuously changing quantities. We show how the use of the first order logic part of OLP allows for a flexible qualitative representation of change, and how the OLP procedure supports a wide range of reasoning tasks on this representation. A second contribution links OLP Event Calculus to temporal knowledge bases, showing how such a knowledge base can be represented as an OLP theory and how the usual tasks of a knowledge base can in principle be performed using the existing procedures for OLP. Moreover we indicate how the OLP representation bridges the gap between the knowledge base and the applications using it (for example for planning), as the same representation is used in either. In addition, we show how also more "knowledge base maintenance" oriented tasks, in particular resolving inconsistency in the knowledge base, are supported by a procedure like the one used for OLP. As is required from a declarative approach, the above results are obtained using only one representation and one general procedure. In a third contribution we use OLP Event Calculus in another entirely different setting for which a set of specialised languages exists: the area of protocol specification. We show how a process protocol can be naturally represented as a dynamic entity in OLP Event Calculus. The representation is of the same size as one produced using specialised protocol specification languages, but is more general and contains more information (and hence can also be used as the basis for other applications, as we indicate). We also show how classical specifications can be mapped to OLP Event Calculus, so that specifications in different languages can be combined into one.

Where in Chapter 6 the flexibility of OLP Event Calculus is shown to allow us to tackle a very wide range of problems, also beyond the setting it was originally intended for, we return in Chapter 7 to classical temporal reasoning in artificial intelligence, in an attempt to tackle the important open problems in that domain. In this context the flexibility of OLP Event Calculus must be held in check by strong guidelines and restrictions to ensure correct behaviour: the formalism has the needed expressive power to deal with the existing problems, but it leaves too much freedom in which the non-expert user may get lost. The restrictions must ensure that a theory represents what is intended, but may on the other hand not limit the expressive power of the formalism. For this reason we develop a language $\mathcal{ER}$ which contains the constructs needed for representing temporal domains, but not the full power of OLP. We show how $\mathcal{ER}$ deals with the major issues in temporal reasoning and compare it with many existing languages in the field. Meanwhile, we address also a lot of interesting side issues. To

close the circle we provide a mapping of $\mathcal{ER}$ theories to a variant of OLP Event Calculus, showing more precisely which part of the latter formalism is retained in $\mathcal{ER}$.

Finally in Chapter 8 we summarise and conclude this thesis.

Most of the contributions of this thesis are adapted from publications. In particular Chapter 3 is based on [114]. A short version of Chapter 5 has been published as [112], and a full version as [113]. The three contributions in Chapter 6 are described in [110], [111] and [29]. A preliminary paper containing the essence of the work in Chapter 7 is published as [115], and a full version is being prepared for journal submission.

Chapter 2

Technical Background

In this chapter we present the syntax and semantics of first order logic, logic programming and open logic programming, and outline the procedure used for reasoning on open logic programs. Our presentation of first order logic and logic programming is largely based on those in [33] and [67], where more examples and proofs can be found. The semantics of open logic programming is that of [26], though our formalisation deviates in some respects. A description of the proof procedure has been adapted from [25]. More on open logic programming in general can be found in [23].

2.1 First Order Logic

We first introduce the syntax and semantics of first order logic.

Definition 2.1.1 (alphabet) *An alphabet consists of the following classes of symbols:*

1. *variables*
2. *function symbols*
3. *predicate symbols*
4. *connectives: "$\neg$" (negation), "$\wedge$" (conjunction), "$\vee$" (disjunction), "$\leftarrow$" (implication) and "$\leftrightarrow$" (equivalence)*
5. *quantifiers: "$\forall$" (universal quantifier) and "$\exists$" (existential quantifier)*
6. *punctuation symbols: "(", ")" and ","*

The first three classes are not fixed: they can have different elements in different alphabets. In general we assume a countably infinite set of variables. These will be denoted by capital letters. Function and predicate symbols will be denoted by lower case letters. As much as possible we will take function symbols from the first half of the Latin alphabet and predicate symbols from the second half. It will usually be clear from the context whether a symbol is a function or predicate symbol.

Both function and predicate symbols have an associated *arity*, a natural number indicating the number of arguments the symbol takes. We will sometimes denote a function or predicate symbol f with arity n as f/n. A function symbol with arity 0 is also called a constant, a predicate symbol with arity 0 a proposition.

Definition 2.1.2 (term) *The set of terms over a given alphabet is defined inductively as follows:*

- *a variable is a term*
- *a constant is a term*
- *a function symbol f with arity $n \geq 0$ applied to a sequence $t_1, \ldots, t_n$ of terms, denoted $f(t_1, \ldots, t_n)$, is a term*

Definition 2.1.3 (atom) *The set of atoms over a given alphabet is defined as follows:*

- *a proposition is an atom*
- *a predicate symbol p with arity $n \geq 0$ applied to a sequence $t_1, \ldots, t_n$ of terms, denoted $p(t_1, \ldots, t_n)$, is an atom*

If $n = 0$, $p(t_1, \ldots, t_n)$ is a different notation for p and $f(t_1, \ldots, t_n)$ for f.

Definition 2.1.4 (well-formed formula) *The set of well-formed formulae over a given alphabet is defined inductively as follows:*

- *an atom is a well-formed formula*
- *if F and G are well-formed formulae, then $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$ and $(F \leftrightarrow G)$ are well-formed formulae.*
- *if X is a variable and F a well-formed formula, then $(\forall X : F)$ and $(\exists X : F)$ are well-formed formulae.*

To simplify notation, punctuation symbols will be omitted where possible. In the absence of punctuation symbols, priority between connectives and quantifiers is as follows, from highest to lowest: first $\neg, \forall, \exists$, then $\wedge$, then $\vee$, then $\leftarrow, \leftrightarrow$.

We adopt the following terminology concerning variables and quantifiers:

Definition 2.1.5 (scope, free variable, groundness, closure) *The scope of $\forall X$ in $(\forall X : F)$ and of $\exists X$ in $(\exists X : F)$ is F.*
A variable X is free in a formula iff it has an occurrence not inside the scope of any quantifier.
A sentence or closed formula is a well-formed formula without free variables.
A ground term (ground formula) is a term (formula) without variables.
The universal closure $\forall(F)$ of a formula F is $\forall X_1 : (\ldots(\forall X_m : F)\ldots)$ where $X_1 \ldots X_m$ are the free variables in F in any order. Similarly the existential closure $\exists(F)$ of F is $\exists X_1 : (\ldots(\exists X_m : F)\ldots)$ with $X_1 \ldots X_m$ the free variables in F.

The semantics of first order logic is defined as follows.

Definition 2.1.6 (interpretation) *An interpretation I consists of*

- *A pre-interpretation I_0, consisting of a domain D and a mapping of n-ary function symbols to n-ary functions $D^n \to D$.*
- *A truth function $\mathcal{H}_I$, mapping n-ary predicate symbols to n-ary relations (i.e. subsets of D^n, or equivalently n-ary functions $D^n \to \{\mathbf{t}, \mathbf{f}\}$).*[1]

Definition 2.1.7 (variable assignment) *A variable assignment α is a function mapping all variables to elements of D. We write $F(X/d)$ to denote the formula obtained from F by assigning the domain element d to the variable X, i.e. replacing all occurrences of X by d.*

Given an interpretation $I =< I_0, \mathcal{H}_I >$ and a variable assignment α, the pre-interpretation I_0 can be extended in a unique way to a mapping $\tilde{I}_\alpha$ of terms to elements of D, and the truth function $\mathcal{H}_I$ to a mapping $\mathcal{H}_{I,\alpha}$ of well-formed formulae to $\mathbf{t}$ or $\mathbf{f}$. We assume an order $\mathbf{t} < \mathbf{f}$ on truth values. Moreover $\mathbf{t}^{-1} = \mathbf{f}$ and $\mathbf{f}^{-1} = \mathbf{t}$.

Definition 2.1.8 (extended pre-interpretation and truth function) *The extended pre-interpretation function $\tilde{I}_\alpha$ is defined as*

[1] If the predicate "=" occurs in the alphabet, it is always interpreted as the identity relation. Moreover we assume the presence of two special propositional symbols *true* and *false* in the alphabet, with $\mathcal{H}_I(true) = \mathbf{t}$ and $\mathcal{H}_I(false) = \mathbf{f}$.

- $\tilde{I}_\alpha(X) = \alpha(X)$ *for any variable* X.
- $\tilde{I}_\alpha(f(t_1, \ldots, t_n)) = I_0(f/n)(\tilde{I}_\alpha(t_1), \ldots, \tilde{I}_\alpha(t_n))$ *for any functor* f *of arity* n *and terms* $t_1, \ldots, t_n$.

The extended truth function $\mathcal{H}_{I,\alpha}$ *is defined as*

- $\mathcal{H}_{I,\alpha}(p(t_1, \ldots, t_n)) = \mathcal{H}_I(p)(\tilde{I}_\alpha(t_1), \ldots, \tilde{I}_\alpha(t_n))$ *for any atom* $p(t_1, \ldots, t_n)$
- $\mathcal{H}_{I,\alpha}(\neg F) = \mathcal{H}_{I,\alpha}(F)^{-1}$
- $\mathcal{H}_{I,\alpha}(F_1 \wedge F_2) = min(\{\mathcal{H}_{I,\alpha}(F_1), \mathcal{H}_{I,\alpha}(F_2)\})$
- $\mathcal{H}_{I,\alpha}(F_1 \vee F_2) = max(\{\mathcal{H}_{I,\alpha}(F_1), \mathcal{H}_{I,\alpha}(F_2)\})$
- $\mathcal{H}_{I,\alpha}(\forall X : F) = min(\{\mathcal{H}_{I,\alpha}(F(X/d)) | d \in D\})$
- $\mathcal{H}_{I,\alpha}(\exists X : F) = max(\{\mathcal{H}_{I,\alpha}(F(X/d)) | d \in D\})$
- $\mathcal{H}_{I,\alpha}(F_1 \leftarrow F_2) = \mathcal{H}_{I,\alpha}(F_1 \vee \neg F_2)$
- $\mathcal{H}_{I,\alpha}(F_1 \leftrightarrow F_2) = \mathcal{H}_{I,\alpha}((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$

Note that for closed formulae, the truth value is independent of the variable assignment α and we can write $\mathcal{H}_I(F)$ instead of $\mathcal{H}_{I,\alpha}(F)$. We write $I \models F$ to denote $\mathcal{H}_I(F) = \mathbf{t}$.

Definition 2.1.9 (model) *A first order logic theory* S *is a set of closed well-formed formulae. An interpretation* I *is a model of* S, *denoted* $I \models S$, *if and only if for all* $F \in S$, $I \models F$. *We say a theory* S *entails a formula* F, *written* $S \models F$, *iff each model of* S *is also a model of* F.

As an example, assume an interpretation I with domain $\{1, 2\}$, in which the predicate symbol p is mapped to the relation $\{(1)\}$ and the constant symbols a and b to 1 and 2 respectively. Then $I \models p(a)$ and $I \models \exists X : p(X)$, but $I \not\models p(b)$ and $I \not\models \forall X : p(X)$. The theory $\{p(a), p(b)\}$ entails $\forall X : p(X)$ as well as $\exists X : p(X)$.

2.2 Logic Programming

Logic programming ([56]) is a logic-based programming and knowledge representation language, for which rather efficient reasoning procedures exist. Syntactically, logic programs are a particular subset of first order logic:

Definition 2.2.1 (clause, logic program) *A literal is an atom A or its negation $\neg A$. A is called a positive literal, $\neg A$ a negative literal.*
A normal clause is a formula $A \leftarrow B_1 \wedge \ldots \wedge B_m$, $m \geq 0$, where A is an atom and $B_1 \ldots B_m$ literals. If $m = 0$ we write the clause as $A \leftarrow true$ or as a shorthand simply as A.
A is called the head of the clause, $B_1 \wedge \ldots \wedge B_m$ the body.
A logic program is a set of normal clauses.
The definition of a predicate p is the set of clauses containing p in the head.

In logic programs, one usually writes a "," rather than "$\wedge$". Moreover all free variables are assumed to be implicitly universally quantified with scope the clause in which they occur (i.e. a clause denotes its universal closure). We adopt these conventions.

Some interesting classes of logic programs are usually distinguished, in particular because their semantics can be defined in a relatively simple way.

Definition 2.2.2 (types of logic programs) *A Horn clause is a normal clause in which all B_i are atoms. A definite program is a program consisting only of Horn clauses.*
We say a predicate p depends on a predicate q if q occurs in p's definition, or if there is a predicate r which occurs in p's definition and which depends on q. p depends positively on q if there is a dependency chain from p to q with only positive occurrences of predicates. p depends negatively on q if there is a dependency chain from p to q with at least one negative predicate occurrence.
A hierarchical program is a program in which no predicate depends on itself.
A stratified program is a program in which no predicate depends negatively on itself.

The semantics of logic programs is often defined in terms of Herbrand models and Herbrand interpretations ([45]): the Herbrand universe is the set of all ground terms over the used alphabet. A Herbrand interpretation is an interpretation with domain the Herbrand universe and in which the pre-interpretation maps each ground term to itself. A Herbrand model is a Herbrand interpretation which is a model. In particular for open logic programs the restriction to Herbrand interpretations is not desirable, which is why we will present more generally applicable semantics.

The semantics of a logic program differs from the corresponding first order logic theory in its incorporation of the closed world assumption. This closed world assumption can not be expressed in first-order logic (its axiomatisation in classical logic requires a second-order logic axiom, like for example those in [71]).

Roughly, the assumption is that only the atoms which are implied by the clauses in the program are true, and all other atoms are false. This intuition can be formalised in different ways for different classes of programs. For hierarchical programs, all existing semantics coincide with the Clark completion semantics ([19]): the meaning of a hierarchical logic program is given by the first order logic theory obtained as follows.

1. Rewrite each clause $p(t_1, \ldots, t_m) \leftarrow Body$, introducing new variables $X_1 \ldots X_m$, as $p(X_1, \ldots, X_m) \leftarrow Body, X_1 = t_1, \ldots, X_m = t_m$.

2. For each predicate p, include in the theory the formula $\forall(p(X_1, \ldots, X_m) \leftrightarrow [Body_1 \vee \ldots \vee Body_n])$, where $Body_1 \ldots Body_n$ are the bodies of all clauses with head $p(X_1, \ldots, X_m)$. If $n = 0$, $[Body_1 \vee \ldots \vee Body_n]$ is the empty disjunction, which we write as the equivalent formula *false*.

3. Add Clark's Free Equality axioms $\mathcal{FEQ}(T)$ ([19]) to the then obtained theory T. These axioms are

   (a) For each functor f/n with $0 \leq n$:
   $\forall(f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \rightarrow X_1 = Y_1 \wedge \ldots \wedge X_n = Y_n)$

   (b) For each pair of different functors f/n and g/m:
   $\forall(\neg f(X_1, \ldots, X_n) = g(Y_1, \ldots, Y_m))$

   (c) For each term t containing a variable X: $\forall(\neg t = X)$.

The Free Equality axioms ensure that two different ground terms always denote different domain elements.

For logic programs in general, the definition of the semantics is more complex and requires a number of additional concepts. The main complication with respect to completion semantics is that the closed world assumption needs to be extended in an intuitive way to programs in which predicates can depend on themselves. The most general semantics, which assigns a unique model to each normal logic program, is the well-founded semantics ([116]). To avoid too many redundant formalisations, we do not present this semantics in detail here, as we will present an extension of it for dealing with open predicates in the next section.

2.3 Open Logic Programming

Open logic programming is an extension of logic programming in which the closed world assumption is only adopted for a subset of the predicates. The idea is that this subset of predicates is defined by a logic program, the

other predicates are left undefined (open), and first order logic sentences may give partial information on these open predicates. Formally, open logic programming is equivalent to abductive logic programming (for a survey, see [52]). However, it interprets this formalism as a knowledge representation language able to deal with incomplete information, rather than as a study of abduction (for example for belief revision) in the context of logic programming. We elaborate on this view on open logic programming as a knowledge representation language in Chapter 3.

Definition 2.3.1 (open logic program) *Syntactically, an open logic program* $T = < P, O, C >$ *consists of*

- P *: A set of normal clauses (a normal logic program).*
- O *: A set of open (abducible, undefined) predicates.*
- C *: A set of general first order logic sentences.*

Open predicates have no definition, i.e. they can not occur in the head of any clauses.[2]

The different existing semantics for open logic programs correspond closely to those for normal logic programs. Intuitively, the models of an open logic program according to a specific semantics are obtained by taking any interpretation for the open predicates, then with this interpretation fixed taking the models of the normal logic program part according to the corresponding normal logic programming semantics, and finally retaining from the resulting models only those in which all sentences of the first order logic part are true.

For hierarchical programs this intuition (extending the Clark completion) is formalised as follows ([20]): the meaning of a hierarchical open logic program $T = < P, O, C >$ is given by the FOL theory obtained in the following way:

1. Rewrite each clause $p(t_1, \ldots, t_m) \leftarrow Body.$, introducing new variables $X_1 \ldots X_m$, as $p(X_1, \ldots, X_m) \leftarrow Body, X_1 = t_1, \ldots, X_m = t_m$.
2. For each predicate $p \notin O$, include in the theory the formula $\forall (p(X_1, \ldots, X_m) \leftrightarrow (Body_1 \vee \ldots \vee Body_n))$, where $Body_1 \ldots Body_n$ are the bodies of all clauses with head $p(X_1, \ldots, X_m)$. If $n = 0$, $[Body_1 \vee \ldots \vee Body_n]$ is the empty disjunction, which we write as the equivalent formula *false*.

[2]Note that it is also possible for a defined predicate not to occur in the head of any clauses: it then has an empty definition.

3. Add Clark's Free Equality axioms to the theory.

4. Add all formulae in C to the theory.

As an example, consider the following open logic program:

$$\begin{array}{rcl} q(a,Y) & \leftarrow & p(Y). \\ r(X,Y) & \leftarrow & \neg q(X,Y). \end{array}$$

with open predicate $p/1$ and FOL axiom $\neg r(a,b)$. The semantics of this program is given by the FOL theory

$$\begin{array}{c} \forall X,Y: q(X,Y) \leftrightarrow p(Y) \wedge X = a \\ \forall X,Y: r(X,Y) \leftrightarrow \neg q(X,Y) \\ \neg r(a,b) \\ a \neq b \end{array}$$

The semantics of normal open logic programs (justification semantics) can be obtained in a similar way as an extension of the well-founded semantics. With respect to completion semantics for open logic programs, the main difference is that the formalisation of the closed world assumption needs to be extended to deal with cyclic dependencies between predicates. Just like for well-founded semantics for normal logic programs, several equivalent formalisations exist. Here we choose a least fixpoint characterisation, which is closest to the inductive definition semantics we will propose in Chapter 7. For the original formalisation and correspondences with other semantics we refer to [26]. Note that the technical discussions in all but Chapter 7 of this text will be based on the completion semantics for open logic programs and will only require a general intuition of the more complex semantics below. In Chapter 7 we will discuss the underlying intuitions of this semantics in more detail.

The following notions are required.

Definition 2.3.2 (3-valued interpretation) *A 3-valued interpretation I consists of*

- *A pre-interpretation I_0, consisting of a domain D and a mapping of n-ary function symbols to n-ary functions $D^n \rightarrow D$.*
- *A truth function $\mathcal{H}_I$, mapping n-ary predicate symbols to n-ary functions $D^n \rightarrow \{\mathrm{t}, \mathrm{u}, \mathrm{f}\}$).*[3]

[3] If the predicate "=" occurs in the alphabet, it is always interpreted as the identity relation. We assume the presence of two special propositional symbols *true*, *undef* and *false* in the alphabet, with $\mathcal{H}_I(true) = \mathrm{t}$, $\mathcal{H}_I(undef) = \mathrm{u}$ and $\mathcal{H}_I(false) = \mathrm{f}$.

The set of 3-valued interpretations will be denoted $\mathcal{J}$. We say a 3-valued interpretation is 2-valued in a certain predicate iff it does not assign **u** to any instance of the predicate.

Given a 3-valued interpretation I and a variable assignment α, I_0 can be extended to a mapping $\tilde{I}_\alpha$ of terms to elements of D and $\mathcal{H}_I$ to a mapping $\mathcal{H}_{I,\alpha}$ of well-formed formulae to **t**, **u** or **f**. This can be done in the same way as for a 2-valued interpretation if we assume the order $\mathbf{t} \leq \mathbf{u} \leq \mathbf{f}$ on truth values and $\mathbf{t}^{-1} = \mathbf{f}$, $\mathbf{u}^{-1} = \mathbf{u}$ and $\mathbf{f}^{-1} = \mathbf{t}$:

Definition 2.3.3 (extended 3-valued interpretation functions) *The extended pre-interpretation function $\tilde{I}_\alpha$ is defined as*

- $\tilde{I}_\alpha(X) = \alpha(X)$ *for any variable* X.
- $\tilde{I}_\alpha(f(t_1, \ldots, t_n)) = I_0(f/n)(\tilde{I}_\alpha(t_1), \ldots, \tilde{I}_\alpha(t_n))$ *for any functor f of arity n and terms $t_1, \ldots, t_n$.*

The extended 3-valued truth function $\mathcal{H}_{I,\alpha}$ is defined as

- $\mathcal{H}_{I,\alpha}(p(t_1, \ldots, t_n)) = \mathcal{H}_{I,\alpha}(p)(\tilde{I}_\alpha(t_1), \ldots, \tilde{I}_\alpha(t_n))$ *for any atom* $p(t_1, \ldots, t_n)$
- $\mathcal{H}_{I,\alpha}(\neg F) = \mathcal{H}_{I,\alpha}(F)^{-1}$
- $\mathcal{H}_{I,\alpha}(F_1 \wedge F_2) = min(\{\mathcal{H}_{I,\alpha}(F_1), \mathcal{H}_{I,\alpha}(F_2)\})$
- $\mathcal{H}_{I,\alpha}(F_1 \vee F_2) = max(\{\mathcal{H}_{I,\alpha}(F_1), \mathcal{H}_{I,\alpha}(F_2)\})$
- $\mathcal{H}_{I,\alpha}(\forall X : F) = min(\{\mathcal{H}_{I,\alpha}(F(X/d)) | d \in D\})$
- $\mathcal{H}_{I,\alpha}(\exists X : F) = max(\{\mathcal{H}_{I,\alpha}(F(X/d)) | d \in D\})$
- $\mathcal{H}_{I,\alpha}(F_1 \leftarrow F_2) = \mathcal{H}_{I,\alpha}(F_1 \vee \neg F_2)$
- $\mathcal{H}_{I,\alpha}(F_1 \leftrightarrow F_2) = \mathcal{H}_{I,\alpha}((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$

For closed formulae, the truth value is independent of the variable assignment α and we write $\mathcal{H}_I(F)$ instead of $\mathcal{H}_{I,\alpha}(F)$. Likewise for ground terms $\tilde{I}_\alpha$ is independent of the variable assignment and is written as $\tilde{I}$. We write $I \models F$ to denote $\mathcal{H}_I(F) = \mathbf{t}$.

Definition 2.3.4 (3-valued model, weak model) *A 3-valued interpretation I is a model of a set of formulae S, denoted $I \models S$, if and only if for all $F \in S$, $\mathcal{H}_I(F) = \mathbf{t}$. I is a weak model of S if for all $F \in S$, $\mathcal{H}_I(F) \geq \mathbf{u}$. Hence, a model is always a weak model but not vice versa.*

Next, we need the concept of a proof tree. For Herbrand interpretations, the following definition is sufficient:

Definition 2.3.5 (proof tree (Herbrand)) *A proof tree Tr for a ground atom p according to a particular open logic program $< P, O, C >$ is a tree of ground literals such that*

- *the root of Tr is p*
- *for each non-leaf node n of Tr with a set of immediate descendants B, n is a defined atom and either $n \leftarrow \bigwedge_{b \in B} b$ is a ground instance of a program clause of P, or $B = \{false\}$.*
- *Tr is maximal, i.e. atoms of defined predicates do not occur in leaf nodes. Each leaf node thus contains true, false, an open atom or a negative literal.*[4]
- *Tr is finite, i.e. contains no infinite branches*

To allow for dealing with non-Herbrand interpretations, we must extend this notion slightly, using the following concepts.

Definition 2.3.6 (D-ground literal/clause) *Given a domain D, a D-ground literal is a ground literal based on $\mathcal{F} \cup D$, where $\mathcal{F}$ is the set of functors in the alphabet. A D-ground instance of a program clause Cl is a formula obtained from Cl by replacing all variables with ground terms based on $\mathcal{F} \cup D$.*

In other words, we add the domain elements as constants to the alphabet and consider ground instances w.r.t. the resulting alphabet. Thus we ensure that domain elements not explicitly mentioned in the theory can exist.[5]

Definition 2.3.7 (evaluated atom) *Given a pre-interpretation I_0, the evaluated atom of a ground atom $p(t_1, \ldots, t_m)$, denoted $\tilde{I}(p(t_1, \ldots, t_m)$, is the atom $p(\tilde{I}(t_1), \ldots, \tilde{I}(t_m))$.*

Definition 2.3.8 (proof tree (non-Herbrand)) *A proof tree Tr for a D-ground atom p according to a particular open logic program $< P, O, C >$ is a tree of D-ground literals such that*

- *the root of Tr is p*
- *for each non-leaf node n of Tr with a set of immediate descendants B, n is a defined atom such that either $\tilde{I}(n) = \tilde{I}(n')$ and $n' \leftarrow \bigwedge_{b \in B} b$ is a D-ground instance of a program clause of P, or $B = \{false\}$.*

[4] Observe that the symbol *undef* should never occur explicitly in a program, and therefore neither in a proof tree.

[5] We assume that any pre-interpretations based on this extended alphabet map each domain element to itself.

- *Tr is maximal, i.e. atoms of defined predicates do not occur in leaf nodes. Each leaf node thus contains true, false, an open atom or a negative literal.*
- *Tr contains no infinite branches*

Given some 3-valued interpretation I which is 2-valued in the open predicates, for each atom p in the language, we define its *supported value* w.r.t. I, denoted $SV_I(p)$, as the truth value proven by its best proof tree, i.e.

- $SV_I(p) = \mathbf{t}$ if p has a proof tree with all leaves containing true facts w.r.t. I;
- $SV_I(p) = \mathbf{f}$ if each proof tree of p has a false fact w.r.t. I in a leaf;
- $SV_I(p) = \mathbf{u}$ otherwise; i.e. if each proof tree of p contains a non-true leaf, and some proof tree contains only non-false leaves.

Note that an open atom has only one proof tree, consisting of only the atom itself, hence $I(p) = SV_I(p)$ for each atom of an open predicate.

The models of an open logic program $< P, O, C >$ are now obtained as follows: take any interpretation I_O which assigns $\mathbf{t}$ or $\mathbf{f}$ to any open atom. $I_{O,\perp}$ is the interpretation which assigns $\mathbf{u}$ to all defined atoms and the same value as I_O to all open atoms. Then construct $I_{O,P}$ as follows.

For a definite program P, $I_{O,P}$ is the interpretation mapping each p to $SV_{I_{O,\perp}}(p)$, i.e. each atom is mapped to its supported value w.r.t. $I_{O,\perp}$. For non-definite programs, $I_{O,P}$ is obtained as a fixpoint of this operation:

Definition 2.3.9 (positive induction operator) *The operator $\mathcal{PI}_{O,P} : \mathcal{J} \rightarrow \mathcal{J} : I \rightarrow I'$ is defined such that $\forall p : I'(p) = SV_I(p)$.*

It can be proven that this operator always has a least fixpoint $\mathcal{PI}_{O,P}\uparrow$ with respect to the specificity order on interpretations, which is the pointwise extension to interpretations of the partial order $\mathbf{t} > \mathbf{u}, \mathbf{f} > \mathbf{u}$, i.e. $I_1 \leq I_2$ iff for all atoms at, $I_1(at) \leq I_2(at)$. In other words $I_1 \leq I_2$ if the set of atoms mapped to $\mathbf{u}$ by I_2 is a subset of the set of atoms mapped to $\mathbf{u}$ by I_1 and I_2 coincides with I_1 on the atoms mapped to $\mathbf{t}$ or $\mathbf{f}$ by I_1. We can then define $I_{O,P}$ as:

Definition 2.3.10 *Given $< P, O, C >$, $I_{O,P} = \mathcal{PI}_{O,P}\uparrow$.*

The least fixpoint of this operator can be proven to coincide with the least fixpoint of the well-founded operator of [116]. Details can be found in [24].

The models of an open logic program $< P, O, C >$ are now all those interpretations $I_{O,P}$ obtained from an interpretation $I_{O,\perp}$ as above, and for which in addition $I_{O,P} \models C$ and $I_{O,P} \models \mathcal{FEQ}(T)$, i.e. the models of the logic program in which in addition all FOL axioms are true and Clark's Free Equality axioms hold for the elements of the Herbrand universe.

As an example, consider again the open logic program

$$\begin{array}{lll} q(a,Y) & \leftarrow & p(Y). \\ r(X,Y) & \leftarrow & \neg q(X,Y). \end{array}$$

with open predicate $p/1$ and FOL axiom $\neg r(a,b)$. Moreover assume the domain only contains two elements a and b. Models of this theory are found as follows. Take any interpretation I_O of $p/1$. Either $I_O(p(b)) = \mathbf{t}$ or $I_O(p(b)) = \mathbf{f}$ and likewise for $p(a)$. For the interpretation which assigns $\mathbf{t}$ to both atoms, we find

$$\begin{array}{l} I_{I_O,\perp}(p(a)) = \mathbf{t} \\ I_{I_O,\perp}(p(b)) = \mathbf{t} \\ \forall X,Y : I_{I_O,\perp}(q(X,Y)) = \mathbf{u} \\ \forall X,Y : I_{I_O,\perp}(r(X,Y)) = \mathbf{u} \end{array}$$

Applying the positive induction operator once then yields

$$\begin{array}{l} I_{I_O,1}(p(a)) = \mathbf{t} \\ I_{I_O,1}(p(b)) = \mathbf{t} \\ \forall Y : I_{I_O,1}(q(a,Y)) = \mathbf{t} \\ \forall Y : I_{I_O,1}(q(b,Y)) = \mathbf{f} \\ \forall X,Y : I_{I_O,1}(r(X,Y)) = \mathbf{u} \end{array}$$

and after a second iteration we find

$$\begin{array}{l} I_{I_O,2}(p(a)) = \mathbf{t} \\ I_{I_O,2}(p(b)) = \mathbf{t} \\ \forall Y : I_{I_O,2}(q(a,Y)) = \mathbf{t} \\ \forall Y : I_{I_O,2}(q(b,Y)) = \mathbf{f} \\ \forall Y : I_{I_O,2}(r(a,Y)) = \mathbf{f} \\ \forall Y : I_{I_O,2}(r(b,Y)) = \mathbf{t} \end{array}$$

which is a fixpoint. The FOL axiom $\neg r(a,b)$ is satisfied, so this interpretation is a model of the open logic program. Similarly we find interpretations based on the other three interpretations of $p/1$. The two interpretations in which $p(b)$ is false do not yield models of the open logic program, as the FOL axiom is violated.

Observe that the two models coincide with those of the completion (for the given domain). This is necessarily so since the program is hierarchical.

As an example of a non-hierarchical program, assume

$$p \leftarrow \neg q.$$
$$q \leftarrow \neg p.$$

with no open predicates or FOL axioms. Here, both predicates depend negatively on each other and hence themselves. Applying the induction operator to $\bot$ yields $\bot$, hence the only model of the theory assigns u to both p and q.

On the other hand, the completion of this program is

$$p \leftrightarrow \neg q.$$
$$q \leftrightarrow \neg p.$$

which has two models $\{p, \neg q\}$ and $\{q, \neg p\}$. The difference between the two semantics is intuitively that justification semantics requires a definition to be constructive. In the above program there is no way to assign a truth value to either p or q without making assumptions about the other predicate. The truth value u indicates this non-constructiveness of the definition.

2.4 Procedural Aspects

From a procedural point of view, an advantage of logic programming over first order logic is that due to the more restricted structure of formulae efficient proof procedures can be implemented by incorporating specific control strategies in the resolution procedure of [91]. For definite programs, such an efficient procedure was given in [56]. This procedure was later called SLD-resolution (Selection rule-driven Linear resolution for Definite programs). It was further extended to deal with negative literals, resulting in SLDNF-resolution (where the NF stands for *negation as failure*, the procedural counterpart of the closed world assumption). For details we refer to [67].

For reasoning on open logic programs this procedure requires an additional extension to be able to deal with open predicates and with first-order logic axioms. This gave rise to the procedure SLDNFA ([25]). We outline the general principles of this procedure as we will need them in Chapters 3 and 6.

We first need the concept of a variable substitution, which is the basis of unification and plays a central role in the procedure.

Definition 2.4.1 (substitutions) *A substitution is a finite set $\{X_1/t_1, \ldots, X_n/t_n\}$ where $X_1 \ldots X_n$ are variables, $t_1 \ldots t_n$ are terms and $X_i \neq t_i$ for all i. The empty substitution is denoted ε.*
Applying a substitution θ to a term or formula F yields $F\theta$, obtained by simultaneously replacing each free occurrence of each X_i in F by t_i. We say that F_1 is an instance of F_2 if $F_1 = F_2\theta$ for some substitution θ.
Substitutions can be applied to sets of expressions: $\{E_1, \ldots, E_m\}\theta$ is defined as $\{E_1\theta, \ldots, E_m\theta\}$. Moreover substitutions can be composed as follows: if $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ and $\sigma = \{Y_1/s_1, \ldots, Y_n/s_m\}$ then the composition $\theta\sigma$ of θ and σ is defined as $\{X_i/t_i\sigma \mid 1 \leq i \leq n \wedge t_i\sigma \neq X_i\} \cup \{Y_i/s_i \mid 1 \leq i \leq m \wedge Y_i \notin \{X_1, \ldots, X_n\}\}$. Note that with this definition of composition, for any expression E and substitutions θ and σ, $(E\theta)\sigma = E(\theta\sigma)$.
A substitution θ is idempotent if $\theta\theta = \theta$. θ is called relevant for a formula or set of formulae F if all variables in θ occur in F.

For more details, properties and examples we refer to [67].

SLDNFA takes as input an open logic program $T = < P, O, C >$ and a formula F which is a conjunction of literals. The procedure then computes a substitution θ and a set of ground open atoms Δ such that

$$T \cup comp(\Delta) \models F\theta$$

Below we describe how this is achieved by SLDNFA.

Like SLD and SLDNF, SLDNFA is based on the principles of resolution and unification. To prove a formula F, SLD-like procedures try to arrive at a contradiction given that $\neg F$ holds. This strategy is based on the observation that $T \models F$ if and only if $T \cup \neg F$ is inconsistent. This point of view is reflected in the fact that goals in the procedure are written as $\leftarrow F$, a shorthand for $false \leftarrow F$ which is equivalent to $\neg F$. The procedure attempts to reduce the formula F to *true*, or more precisely it attempts to derive the empty goal $\leftarrow true$ (which is a contradiction) from the assumption $\leftarrow F$.

To define this more precisely we need the concepts of unification and resolution. Unification is based on the concept of variable substitution defined above.

Definition 2.4.2 (unifier) *A substitution θ is a unifier of a set of expressions S if $S\theta$ is a singleton. θ is a most general unifier of S iff for each unifier σ of S there exists a substitution γ such that $\sigma = \theta\gamma$.*

Resolution is defined as follows:

Definition 2.4.3 (resolution) *Given a goal $G = \leftarrow L_1, \ldots, L_m$ and a program clause $Cl = A \leftarrow B_1, \ldots, B_n$, with $m \geq 1$ and $n \geq 0$ such that G and Cl have no variables in common (if there are common variables, they are renamed in one of the formulae; this renaming is called* standardising apart*). Then G' is derived by resolution from G and Cl using θ iff*

- *L_k is an atom in G, called the selected atom.*
- *θ is an idempotent and relevant most general unifier of $\{L_k, A\}$.*
- *G' is the goal $(\leftarrow L_1, \ldots, L_{k-1}, B_1, \ldots, B_n, L_{k+1}, \ldots, L_m)\theta$.*

Now, say $G = \leftarrow F$ and $G' = \leftarrow F'$. From the construction it follows that if a theory containing the clause Cl entails F', then it entails $F\theta$. For definite logic programs P the SLD theorem prover tries to apply a sequence of resolution steps (called a derivation) to a goal G such that the goal $\leftarrow true$ is obtained. If there is such a derivation with θ the composition of the unifiers used in it, then $P \models F\theta$. Otherwise $P \models \neg F$.

For normal logic programs the procedure becomes more complex. Resolution can only remove defined atoms from a goal, so another mechanism is required to deal with defined negative literals, leading to SLDNF. This mechanism is the following: from a goal $\leftarrow F = \leftarrow L_1, \ldots, L_m$ a ground negative literal $L_k = \neg l$ can be selected.[6] Then one attempts to construct a complete derivation tree for $\leftarrow l$, i.e. a maximal tree in which each node is obtained by resolution on its parent node and a program clause. The goals in this tree are called negative goals. If this complete derivation tree is finitely failed, i.e. is finite and does not contain $\leftarrow true$ in a node, then $P \models \neg l$. Hence, $\leftarrow F' = \leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_m$ can be obtained from $\leftarrow F$, with $P \models F$ iff $P \models F'$.

The procedure for open logic programs, SLDNFA, needs to deal in addition with open predicates: it constructs a set Δ of open atoms such that these atoms together with the open logic program entail F. This is done by abductive inference steps: in a goal $\leftarrow F = \leftarrow L_1, \ldots, L_m$ a ground open atom L_k can be selected. Then Δ is updated: $\Delta' = \Delta \cup L_k$, and $\leftarrow F' = \leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_m$. If then $P \cup \Delta' \models F'$, then $P \cup \Delta' \models F$.

Given these inference steps, two problems remain to be solved. These are the treatment of non-ground open atoms and of negative open literals. If a non-ground open atom L_k is selected, it is *skolemised*: a substitution σ is applied which binds each variable in L_k to a separate new skolem constant, i.e. a constant which does not yet occur in the theory or the

[6] In general it is not allowed to select non-ground negative literals in SLDNF; as a result the procedure is not complete.

goal. Then $L_k\sigma$ is a ground atom, and we define $\Delta' = \Delta \cup L_k\sigma$ and $G' = (\leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_m)\sigma$. The skolem constants introduced above differ from "normal" constants in that the Free Equality axioms do not apply to them, i.e. a skolem constant may denote the same domain element as a different (skolem or non-skolem) constant or term. As a result, the concepts of substitution and unification need to be extended, since we want atoms that can be equal in an interpretation to be unifiable. The formal details follow below.

Negative open literals pose an additional problem. Such a literal $\neg L$ is entailed by $comp(\Delta)$ as long as there is no $L' \in \Delta$ which unifies with L. However, throughout the derivation Δ tends to grow, and there is no guarantee that if at a particular step in the derivation there is no $L' \in \Delta$ which unifies with L, this will also remain the case throughout the derivation. Hence, $\neg L$ can not be safely considered entailed until Δ is completely known, i.e. at the end of the derivation. More generally, if an atom L occurs in a negative goal, it is possible that the derivation of this goal fails given the current Δ since L can not be resolved, but that L unifies later with an atom in an extended Δ and the derivation can be successfully completed. The assumption $\neg L$ would then be erroneous.

To cure this problem, negative goals $\leftarrow L_1, \ldots, L_m$ containing an open atom L_k need to be remembered and checked again each time a new atom is added to Δ. If at any time L_k unifies with an atom in Δ, say with unifier σ, an additional negative goal $(\leftarrow L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_m)\sigma$ needs to be generated, for which all derivations must fail.

This briefly summarises the issues SLDNFA needs to deal with. We now proceed with the formal details.

Definition 2.4.4 (equality set) *Assume $\mathcal{L}$ is an alphabet and SK a set of skolem constants. An equality set based on $\mathcal{L} \cup SK$ is the set $\{\Box\}$ or a finite set $\{s_1 = t_1, \ldots, s_n = t_n\}$ where s_i, t_i are terms based on $\mathcal{L} \cup SK$.*
An equality set based on $\mathcal{L} \cup SK$ is in solved form if and only if it is either the set $\{\Box\}$ or if each s_i is a variable or a skolem constant, no s_i occurs in any t_j, and t_i is not a variable if s_i is a skolem constant. The set $\{\Box\}$ is called inconsistent, other equality sets in solved form are called consistent.

A set of rewriting rules for reducing an equality set to an equivalent solved form can be found in [23].

Using the above definition we extend the notion of substitution to allow skolem constants to appear in the substitution's domain.

Definition 2.4.5 (extended substitution) *Each consistent equality set $E = \{s_1 = t_1, \ldots, s_n = t_n\}$ in solved form defines a unique substitution $\sigma_E = \{s_1/t_1, \ldots, s_n/t_n\}$. Applying such a substitution to a term or formula*

is defined as before, treating skolem constants like variables. To simplify notation we will from now on use equality sets in solved form as substitutions.

Based on this extended notion of substitution, we can define extended notions of unification and resolution.

Definition 2.4.6 (skolemising substitution) *A skolemising substitution θ of a term, formula, set of terms or set of formulae with free variables $X_1 \dots X_m$ is a substitution $\{X_1/sk_1, \dots, X_m/sk_m\}$ where all sk_i are distinct skolem constants.*

Definition 2.4.7 (positive unification) *A positive unifier of a set S is a substitution $E\theta_{sk}$, where E is a unifier of S and θ_{sk} is a skolemising substitution for the terms which are assigned to skolem constants in E. We say the elements of S positively unify with substitution $E\theta_{sk}$.*

Definition 2.4.8 (positive resolution) *Given a goal $G = \leftarrow L_1, \dots, L_k, \dots, L_m$ with L_k an atom, and a normal clause $Cl = A \leftarrow B_1, \dots, B_n$ sharing no variables with G. G' is derived from G and Cl by positive resolution on L_k with unifier θ if the following conditions hold:*

- *θ is a positive unifier of L_k and A*
- *$G' = (\leftarrow L_1, \dots, L_{k-1}, B_1, \dots, B_n, L_{k+1}, \dots, L_m)\theta$*

Note that if L_k does not contain skolem constants, then positive resolution is just classical resolution.

Definition 2.4.9 (negative unification) *A negative unifier E of a set S is a unifier of S. Let θ be the part of E with variables at the left and E_r the part with skolems at the left. We say the elements of S negatively unify with substitution θ and residue E_r.*

Definition 2.4.10 (negative resolution) *Given a goal $G = \leftarrow L_1, \dots, L_k, \dots, L_m$ with L_k an atom, and a normal clause $Cl = A \leftarrow B_1, \dots, B_n$ sharing no variables with G. G' is derived from G and Cl by negative resolution on L_k if the following conditions hold:*

- *L_k and A negatively unify with variable substitution θ and residue $\{sk_1 = t_1, \dots, sk_l = t_l\}$.*
- *$G' =$
  $(\leftarrow L_1, \dots, L_{k-1}, sk_1 = t_1, \dots, sk_l = t_l, B_1, \dots, B_n, L_{k+1}, \dots, L_m)\theta$*

Note that if L_k does not contain skolem constants, then negative resolution is just classical resolution.

Using these concepts, we can define how SLDNFA works. Our formalisation deviates slightly from the one in [25], in particular in that we group several steps into one, but the result is equivalent. The procedure maintains three sets: one of positive goals PG, one of negative goals NG and one of abduced atoms Δ. Initially NG and Δ are empty and PG contains the input goal $\leftarrow F$. At each step one non-empty goal and one literal in this goal are selected. If a goal from PG is selected, there are the following possibilities:

- the selected literal is a defined atom L or an equality atom. In this case positive resolution is applied with some clause of the program (in case of an equality atom with the reflexivity clause $X = X \leftarrow true$.) and the new goal replaces the selected one; the used substitution θ is applied to PG, NG and Δ.
- the selected literal is a negative literal $\neg L$. In this case the goal L is added to NG, and $\neg L$ is removed from the positive goal.
- the selected literal is an open atom L. The atom is skolemised using a skolemising substitution σ. $L\sigma$ is added to Δ, and for each goal G in NG which has an atom negatively unifiable with $L\sigma$, the goal G' obtained by negative resolution on G and $L\sigma$ is added to NG. σ is applied to PG, NG and Δ.

If a goal from NG is selected, the following possibilities exist:

- the selected literal is a defined atom L or an equality atom. Negative resolution is applied with all clauses $Cl_1, \ldots, Cl_n$ of the program of which the head is unifiable with L (in case of an equality atom with the reflexivity clause $X = X \leftarrow true$.) For each clause, the result of the resolution is added to NG, and the original goal is removed.
- the selected literal is a negative literal $\neg L$. There are two possibilities to be chosen from: either $\leftarrow L$ is added to PG and the negative goal is removed from NG, or $\neg L$ is removed from the negative goal and $\leftarrow L$ is added to NG. The possibility to choose arises from the two ways in which the negative goal may fail: by failure of its selected literal or by success of the selected literal and failure of the rest of the goal.
- the selected literal is an open atom L. Negative resolution is applied with each atom in Δ which negatively unifies with L, and the resulting goals are added to NG. The original negative goal is also retained.

The procedure terminates successfully if PG contains only empty goals and each goal in NG contains an irreducible equality atom. In that case $P \cup comp(\Delta) \models F\theta$. A derivation fails if any goal in NG becomes empty, or if no more selections are possible when a goal in PG is not yet empty. Note that, like in SLDNF, to ensure correctness of the procedure the selection rule needs to be safe, i.e. no non-ground negative literals may be selected. Negative literals containing skolem constants may be selected.

Throughout this discussion we have ignored one issue, namely the treatment of general first-order logic axioms. Nowhere does the procedure take such axioms into account. To deal with general axioms, we need to transform them into a form which can be used by SLDNFA. In practice this is done by writing each axiom A as a general clause $invalid \leftarrow \neg A$ and by presenting the goal $\leftarrow F \wedge \neg invalid$ to SLDNFA when trying to solve $\leftarrow F$. Of course $invalid \leftarrow \neg A$ is not necessarily a normal clause as A can be an arbitrary formula. However, there exists a simple automated technique for transforming a set of general clauses like the above one into an equivalent set of normal clauses. The technique is described in [68], and we summarise it in Appendix A. More details can also be found in Chapters 3 and 6, where we will explicitly use the technique. From now on we will make abstraction of this transformation step except when going into procedural details. In other words, whenever we use SLDNFA for reasoning, we assume that any FOL axioms are transformed into a definition of $invalid$, and that any goal is extended with the conjunct $\neg invalid$.

As an example of SLDNFA-execution, take again the open logic program

$$\begin{array}{lcl} q(a,Y) & \leftarrow & p(Y). \\ r(X,Y) & \leftarrow & \neg q(X,Y). \end{array}$$

with open predicate $p/1$ and FOL axiom $\neg r(a,b)$. The FOL axiom can be written as a definition of $invalid$:

$$invalid \leftarrow r(a,b).$$

Assume we want to check if $\exists Y : p(Y), r(b,Y)$ is possible. We use the goal $\leftarrow p(Y), r(b,Y), \neg invalid$. Then

$PG_0 = \{\leftarrow p(Y), r(b,Y), \neg invalid\}$
$NG_0 = \Delta_0 = \{\}$
Abduction yields
$PG_1 = \{\leftarrow r(b,sk), \neg invalid\}$
$NG_1 = \{\}$
$\Delta_1 = \{p(sk)\}$

where $Y = sk$ is used as skolemising substitution.
Then positive resolution on the first goal results in
$PG_2 = \{\leftarrow \neg q(b, sk), \neg invalid\}$
$NG_2 = \{\}$
$\Delta_2 = \{p(sk)\}$
Then we can move both conjuncts of the positive goal to NG, yielding
$PG_4 = \{\leftarrow true\}$
$NG_4 = \{\leftarrow q(b, sk)\ ,\ \leftarrow invalid\}$
$\Delta_4 = \{p(sk)\}$
Negative resolution on the first negative goal yields no resolvents, hence
$PG_5 = \{\leftarrow true\}$
$NG_5 = \{\leftarrow invalid\}$
$\Delta_5 = \{p(sk)\}$
and two applications of negative resolution on the second goal yield
$PG_6 = \{\leftarrow true\}$
$NG_6 = \{\leftarrow r(a, b)\}$
$\Delta_6 = \{p(sk)\}$
and
$PG_7 = \{\leftarrow true\}$
$NG_7 = \{\leftarrow \neg q(a, b)\}$
$\Delta_7 = \{p(sk)\}$
The negation of this goal can then be moved back to PG:
$PG_8 = \{\leftarrow q(a, b)\}$
$NG_8 = \{\}$
$\Delta_8 = \{p(sk)\}$
whence positive resolution results in
$PG_9 = \{\leftarrow p(b)\}$
$NG_9 = \{\}$
$\Delta_9 = \{p(sk)\}$
Finally, a second abduction step leads to
$PG_{10} = \{\leftarrow true\}$
$NG_{10} = \{\}$
$\Delta_{10} = \{p(sk), p(b)\}$

at which time the procedure terminates successfully. We obtain the answer that $T \cup comp(\{p(sk), p(b)\}) \models (p(Y), r(b, Y))\{Y = sk\}$

In conclusion of this discussion of SLDNFA, we want to point out that it can be used to solve various types of problems. In this respect it differs from procedures for normal logic programs, which can only be used for deductive tasks: since normal logic programs encode complete knowledge on a problem domain, the only interesting reasoning paradigm on them is deduction. Open logic programs can represent partially specified domains,

and on such domains also reasoning paradigms like abduction and model generation, which essentially try to make sensible and useful assumptions about the unknown part of the domain, are of interest. As a result a much wider range of specific tasks can be tackled. SLDNFA is suited for both deductive and abductive forms of reasoning, including model generation. We will illustrate and discuss the suitability of SLDNFA for different forms of reasoning further throughout the thesis, in particular in the context of temporal reasoning.

Chapter 3

A Formal Correspondence between Open Logic Programming and Description Logics

3.1 Introduction

In this chapter we formally investigate the relation between open logic programming and a class of specialised knowledge representation languages called description logics (DL) or concept languages. A first goal of this analysis is to clarify the long-questioned relations between these two related areas of research: though they have been combined in major basic research efforts to enhance the state of the art in computational logic (in the Esprit BRA-projects Compulog and CompulogII), the relation between them has always remained unclear.[1] In the context of this thesis, the analysis has the additional goal to illustrate how OLP, unlike FOL or LP, is perfectly suited as a knowledge representation language, fulfilling the essential requirements of such languages. Moreover, the clear mapping between OLP and description logics allows to highlight the gain in knowledge representation power that OLP offers over these logics.

We start by presenting the ideas underlying description logics and for-

[1] In [18] *C − Logic*, a logic for representing complex objects which is somewhat reminiscent of DLs (in containing class symbols and set-valued labels), has been mapped to FOL. This work is related to some aspects of the work presented in this chapter.

mally defining their syntax and semantics in the following two sections. In section 3.4 we provide a mapping of description logic theories into open logic programs and indicate to which sublanguages of OLP different description logics correspond. Section 3.5 discusses correspondences between algorithms used for reasoning on description logics and the SLDNFA procedure for open logic programs. We conclude with a discussion in section 3.6.

3.2 The Ideas behind Description Logics

At the basis of research on description logics ([47], [48], [30], [15], [17], [21], [5]) lies the idea in [13] and [11] that an expert system — and a knowledge representation language in general — needs to deal with two essentially different kinds of information: on the one hand so-called *assertional* information about the world, and on the other hand *definitional* or *terminological* information.

Assertional information consists of specific observations about actual objects in the world. Examples are the observations that Tom is the father of John or that Mary owns a blue car. This type of assertions can be nicely represented in first-order logic, as a set of general formulae about predicates.

Definitional or terminological information is information about fixed relations between these predicates: it defines concepts (predicates) in terms of each other. As an example, terminological information is that a father is a parent who is male. This kind of information is naturally represented in semantic networks ([87], [9]) and frame-based languages ([76]).

The observation that no existing formalism was powerful enough to represent both types of information (frame-based systems have severely limited assertional power, while FOL is not suited for representing definitions), led to the development of hybrid knowledge representation systems like KRYPTON ([11], [10], [12]). In KRYPTON the knowledge base consists of a terminological part (the *T-Box*), which uses a frame-based language, and an assertional part (the *A-Box*), which uses FOL as a representation language.

This idea of a distinction between A-Box and T-Box is further explored in current description logics (see for example [47], [21]). The T-Box consists of a set of concept definitions in a limited concept definition language. The A-Box contains assertions about concept membership of individual objects. Current research (for example [48], [30], [5]) is mostly concerned with determining definition languages that combine sufficient expressive power with limited algorithmical complexity.

The general idea behind the analysis in this chapter is that an open

logic program $< P, O, C >$ can be interpreted as consisting of a T-Box (the set of clauses P) containing definitions of concepts (predicates) and an A-Box (the set of FOL axioms C) containing general assertional information. In fact, DL concept definitions can be straightforwardly mapped to clauses and A-Box formulae to FOL axioms. An interpretation of a description logic theory corresponds to exactly one interpretation of an open logic program. Under a suitable semantics, the models of the DL theory are the same as those of the corresponding open logic program. The logic programs we study in this chapter are mainly hierarchical programs. For this kind of programs, all semantics coincide with the completion semantics of [20], which is easy to handle. However, we also indicate how the correspondence extends to recent, more expressive description logics which introduce constructs going beyond first order logic.

3.3 Syntax and Semantics of Description Logics

In a description logic, three types of symbols are used: *concepts, roles* and *variables*. Variables range over a set called the domain. Concepts are interpreted as subsets of this domain, while roles are binary relations between domain elements. We will use $A, B, C \ldots$ as concept symbols, $R, S, \ldots$ as role symbols and $x, y, \ldots$ as variable symbols. Also, $\top$ will be used as a special concept symbol representing the entire domain.

A DL theory consists of a terminological (T-Box) and an assertional (A-Box) component. We first describe the language of the T-Box. Several different T-Box languages using varying sets of basic operators (resulting in different complexity results for the algorithms) have been studied in the literature. For our formal comparison, we choose a moderately expressive language, $\mathcal{ALCN}$, as described in [47]. Along the way we indicate straightforward extensions of our analysis to more expressive languages.

The T-Box in a DL theory consists of a number of concept definitions $C == F$, where C is a concept symbol and F a concept description, such that no concept depends on itself.[2] Given that R is a role, C a concept symbol and F, G concept descriptions, valid concept descriptions are the terms:

$$C \mid \forall R : F \mid \exists R : F \mid F \sqcap G \mid F \sqcup G \mid \neg F \mid \; \leq nR \mid \; \geq nR$$

[2] We say C depends on C' if C' is in the definition of C or if C depends on some C'' such that C' is in the definition of C''.

For example,

$$male \sqcap \leq 3child \sqcap [\forall child.male \sqcup \forall child.female]$$

describes the set of men with at most three children, which are either all boys or all girls.

The A-Box consists of a set of *object descriptions* $a : A$ and *relation descriptions* aRb, where a, b are objects, A is a concept symbol and R a role. As an example, $a\ child\ b$ asserts that b is a child of a. From now on we will use the term *(A-Box) constraints* for object and relation descriptions.

Definition 3.3.1 (DL theory) *A DL theory θ consists of a T-Box (denoted $tbox(\theta)$) and an A-Box (denoted $abox(\theta)$). The T-Box is a set of concept definitions $C == F$ such that no concept symbol depends on itself. The A-Box is a set of constraints. A defined concept of θ is any concept symbol C such that $C == F \in tbox(\theta)$. Any other concept symbol is called primitive.*

The semantics of a DL theory is defined via the following concepts:

Definition 3.3.2 (DL interpretation) *An interpretation $\mathcal{I} = (D^{\mathcal{I}}, \mathcal{I}[.])$ consists of a set $D^{\mathcal{I}}$ (the* domain *of $\mathcal{I}$) and a function $\mathcal{I}[.]$ (the* interpretation function *of $\mathcal{I}$) that maps every concept symbol to a subset of $D^{\mathcal{I}}$, every role symbol to a subset of $D^{\mathcal{I}} \times D^{\mathcal{I}}$, and every object to an element of $D^{\mathcal{I}}$ such that $\mathcal{I}[a] \neq \mathcal{I}[b]$ if $a \neq b$ (a unique names assumption on objects: note the correspondence with Clark's Free Equality axioms). Moreover, $\mathcal{I}$ satisfies:*

$$\begin{array}{l}
\mathcal{I}[\top] = D^{\mathcal{I}} \\
\mathcal{I}[\forall R : F] = \{a \in D^{\mathcal{I}} | \forall (a, b) \in \mathcal{I}[R] : b \in \mathcal{I}[F]\} \\
\mathcal{I}[\exists R : F] = \{a \in D^{\mathcal{I}} | \exists (a, b) \in \mathcal{I}[R] : b \in \mathcal{I}[F]\} \\
\mathcal{I}[F \sqcap G] = \mathcal{I}[F] \cap \mathcal{I}[G] \\
\mathcal{I}[F \sqcup G] = \mathcal{I}[F] \cup \mathcal{I}[G] \\
\mathcal{I}[\neg F] = (D^{\mathcal{I}} \setminus \mathcal{I}[F]) \\
\mathcal{I}[\leq nR] = \{a \in D^{\mathcal{I}} | \#(\{b | (a, b) \in \mathcal{I}[R]\}) \leq n\} \\
\mathcal{I}[\geq nR] = \{a \in D^{\mathcal{I}} | \#(\{b | (a, b) \in \mathcal{I}[R]\}) \geq n\}
\end{array}$$

There are some differences in terminology and use of semantical concepts between DL and other logics. In DL, it is uncommon to define the notion of model of a theory. Rather, the following definitions are provided:

Definition 3.3.3 (consistency,subsumption,equivalence) *A concept description F is consistent if there exists an interpretation $\mathcal{I}$ such that $\mathcal{I}[F]$ is nonempty. A concept description F subsumes concept description G if for all $\mathcal{I}$: $\mathcal{I}[G] \subseteq \mathcal{I}[F]$. F and G are equivalent ($F == G$) if for all $\mathcal{I}$: $\mathcal{I}[F] = \mathcal{I}[G]$.*

We intend to map DL theories into OLP and will prove that this is an equivalence preserving operation. To this end, we introduce the concept of a model of a DL theory.

Definition 3.3.4 (DL model) *An interpretation $\mathcal{I}$ is a model of θ iff for all $C == F \in tbox(\theta)$: $\mathcal{I}(C) = \mathcal{I}(F)$ and for all constraints $F \in abox(\theta)$, $\mathcal{I} \models F$.*

3.4 Mapping Description Logic Theories to OLP

Having defined syntax and semantics of description logics, we are ready to devise a mapping from DL to OLP. It can be shown that each model of the resulting open logic program corresponds to exactly one model of the DL theory, and vice versa.

3.4.1 Mapping Function

First of all, we establish the classical correspondence between concepts and roles on the one hand, and predicates on the other hand. A concept C corresponds to a unary predicate $C'/1$, a role R to a binary predicate $R'/2$. For notational convenience we will use the same symbol to denote a concept or role and its corresponding predicate.

We now define a mapping T transforming a DL theory into an open logic program. We can map A-Box constraints to FOL formulae as follows:

$$T(a:C) = C(a). \qquad T(aRb) = R(a,b).$$

As an example, the A-Box $\{Mary : woman;\ Mary\ child\ John\}$ would be mapped to FOL axioms $woman(Mary)$ and $child(Mary, John)$.

Next we map concept definitions to program clauses. This is done in two steps. First we translate the definitions into *general clauses* ([67]). A general clause has the form $A \leftarrow W$ where A is an atom and W an arbitrary FOL expression. In a second step, we use the transformation of Lloyd and Topor ([68] and see appendix A) to transform a set of general clauses into a set of normal clauses.

Assume we have a concept definition $C == F$. This definition is translated into a *general clause* $T(C == F) = C(X) \leftarrow T'(F, X)$. where

$T'(F, X)$ is defined recursively as follows:

$$
\begin{array}{lcl}
T'(C, X) & = & C(X) \;\; (C = \textit{concept symbol}) \\
T'(F_1 \sqcap F_2, X) & = & T'(F_1, X) \wedge T'(F_2, X) \\
T'(F_1 \sqcup F_2, X) & = & T'(F_1, X) \vee T'(F_2, X) \\
T'(\neg F, X) & = & \neg T'(F, X) \\
T'(\forall R : F, X) & = & \forall Y : (R(X, Y) \rightarrow T'(F, Y)) \\
T'(\exists R : F, X) & = & \exists Y : (R(X, Y) \wedge T'(F, Y)) \\
T'(\geq n\, R, X) & = & \exists X_1 \ldots X_n : R(X, X_1) \wedge \ldots \wedge R(X, X_n) \\
 & & \wedge (X_1 \neq X_2) \wedge (X_1 \neq X_3) \wedge \ldots \wedge (X_{n-1} \neq X_n) \\
T'(\leq n\, R, X) & = & \forall X_1 \ldots X_{n+1} : (R(X, X_1) \wedge \ldots \wedge R(X, X_{n+1})) \\
 & & \rightarrow ((X_1 = X_2) \vee (X_1 = X_3) \vee \ldots \vee (X_n = X_{n+1}))
\end{array}
$$

Note that primitive concepts have no definition in the DL theory. Likewise they have no defining clauses in the general open logic program. They correspond to open predicates.

Recently description logics have been extended with other, more expressive constructs, in particular qualified number restrictions on concepts and various role constructors like inverse, intersection, union, concatenation and reflexive transitive closure. The function T' can easily be extended to deal with these: a qualified number restriction $\geq n\, R.C$ or $\leq n\, R.C$ is translated exactly like its unqualified equivalent above, with each instance of $R(X, Y)$ replaced by $R(X, Y) \wedge T'(C, Y)$. As for non-primitive roles, these require T' to be extended explicitly to roles instead of always mapping R to $R(X, Y)$:

$$
\begin{array}{lcl}
T'(R, (X, Y)) & = & R(X, Y) \;\; (R \textit{ primitive}) \\
T'(R^-, (X, Y)) & = & T'(R, (Y, X)) \\
T'(R_1 \sqcap R_2, (X, Y)) & = & T'(R_1, (X, Y)) \wedge T'(R_2, (X, Y)) \\
T'(R_1 \sqcup R_2, (X, Y)) & = & T'(R_1, (X, Y)) \vee T'(R_2, (X, Y)) \\
T'(R_1 \circ R_2, (X, Y)) & = & T'(R_1, (X, Z)) \wedge T'(R_2, (Z, Y))
\end{array}
$$

The reflexive transitive closure R^* of a role R is a more interesting extension, as it cannot be expressed in first order logic. In OLP, it can be expressed under justification semantics as follows: all occurrences of R^* are translated to $R^*(X, Y)$, with R^* a new predicate symbol which is defined by the clauses

$$
\begin{array}{lcl}
R^*(X, Y) & \leftarrow & T'(R, (X, Y)). \\
R^*(X, Y) & \leftarrow & T'(R, (X, Z)) \wedge R^*(Z, Y).
\end{array}
$$

In recent work on description logics (see for example [17]), eliminating the acyclicity condition on concept definitions has been considered, though

there is no agreement yet on the preferable semantics for cycles. Given the results discussed in this thesis and other recent work on OLP, we argue that the semantics offering most expressive power is the one which corresponds to the stronger OLP semantics: we suggest choosing a form of least fixpoint semantics.

As an example of the described mapping, consider the following concept theory:

$$\begin{array}{lcl} C & == & (\exists child.D) \sqcap (\forall child.E) \\ D & == & male \sqcup \geq 3\ child \\ E & == & \leq 1\ child \end{array}$$

with primitive concepts *child* and *male*. This theory is mapped to

$$\begin{array}{lcl} C(X) & \leftarrow & \exists Y : (child(X,Y) \land D(Y)),\ \forall Z : (child(X,Z) \rightarrow E(Z)) \\ D(X) & \leftarrow & male(X) \lor \\ & & child(X,Y), child(X,Z), child(X,U), Y \neq Z, Y \neq U, Z \neq U \\ E(X) & \leftarrow & \forall Y, Z : ([child(X,Y) \land child(X,Z)] \rightarrow Y = Z) \end{array}$$

with open predicates *child*/2 and *male*/1.

The mapping turns any $\mathcal{ALCN}$ description into a general logic program as defined in ([67]), augmented with undefined predicates. In [68] a transformation is described which transforms a general logic program into an equivalent normal logic program. It is proven sound and complete with respect to the completion semantics of [19]. The transformation applies as well to general logic programs augmented with undefined predicates, with respect to the completion semantics for open logic programs ([20]). The details of the transformation are included in Appendix A.

Using this transformation the above program is mapped into the following open logic program (with undefined predicates *child*/2 and *male*/1):

$$\begin{array}{lcl} C(X) & \leftarrow & child(X,Y), D(Y), \neg H(X). \\ H(X) & \leftarrow & child(X,Z), \neg E(Z)). \\ D(X) & \leftarrow & male(X). \\ D(X) & \leftarrow & child(X,Y), child(X,Z), child(X,U), Y \neq Z, Y \neq U, Z \neq U. \\ E(X) & \leftarrow & \neg G(X). \\ G(X) & \leftarrow & child(X,Y), child(X,Z), Y \neq Z. \end{array}$$

As the correctness of the transformation from general into normal (open) logic program is proven in [68], it suffices to prove an equivalence between the concept theory and the general open logic program it is mapped to.

To this end we first define a mapping s from DL interpretations to FOL interpretations, as follows: given $\mathcal{I}$, $s(\mathcal{I})$ is the interpretation with domain

$D^{\mathcal{I}}$ such that for every concept C, every role R and every $X, Y \in D^{\mathcal{I}}$:

$$s(\mathcal{I})(C(X)) = true \leftrightarrow X \in \mathcal{I}[C]$$
$$s(\mathcal{I})(R(X,Y)) = true \leftrightarrow (X,Y) \in \mathcal{I}[R]$$

[3] Then we can prove the following theorem for $\mathcal{ALCN}$ theories θ:

Theorem 3.4.1 *For every interpretation $\mathcal{I}$ of θ, $\mathcal{I}$ is a model of θ if and only if $s(\mathcal{I})$ is a model of $T(\theta)$.*

To help prove this theorem, we first prove an important lemma

Lemma 3.4.1

$$\forall X \in D^{\mathcal{I}} : \mathcal{I}'(T'(F,X)) = true \leftrightarrow X \in \mathcal{I}[F].$$

Proof:
The proof of the lemma is by structural induction on F:

1. $F =$ *concept symbol* : Since $T'(F,X) = F(X)$, the equivalence holds by definition of $s(\mathcal{I})$.

2. $F = F_1 \sqcap F_2$:
   Assume $\mathcal{I}'(T'(F_1,X)) = true \leftrightarrow X \in \mathcal{I}[F_1]$,
   and $\mathcal{I}'(T'(F_2,X)) = true \leftrightarrow X \in \mathcal{I}[F_2]$.
   $T'(F,X) = T'(F_1,X) \wedge T'(F_2,X)$, so
   $\mathcal{I}'(T'(F,X)) = true \leftrightarrow \mathcal{I}'(T'(F_1,X)) = true \wedge \mathcal{I}'(T'(F_2,X)) = true$
   $\leftrightarrow X \in \mathcal{I}[F_1] \wedge X \in \mathcal{I}[F_2] \leftrightarrow X \in \mathcal{I}[F_1 \sqcap F_2] \leftrightarrow X \in \mathcal{I}[F]$

3. $F = F_1 \sqcup F_2$:
   Assume $\mathcal{I}'(T'(F_1,X)) = true \leftrightarrow X \in \mathcal{I}[F_1]$,
   and $\mathcal{I}'(T'(F_2,X)) = true \leftrightarrow X \in \mathcal{I}[F_2]$.
   $T'(F,X) = T'(F_1,X) \vee T'(F_2,X)$, so
   $\mathcal{I}'(T'(F,X)) = true \leftrightarrow \mathcal{I}'(T'(F_1,X)) = true \vee \mathcal{I}'(T'(F_2,X)) = true$
   $\leftrightarrow X \in \mathcal{I}[F_1] \vee X \in \mathcal{I}[F_2] \leftrightarrow X \in \mathcal{I}[F_1 \sqcup F_2] \leftrightarrow X \in \mathcal{I}[F]$

4. $F = \neg F'$:
   Assume $\mathcal{I}'(T'(F',X)) = true \leftrightarrow X \in \mathcal{I}[F']$.
   $T'(F,X) = \neg T'(F',X)$, so
   $\mathcal{I}'(T'(F,X)) = true \leftrightarrow \mathcal{I}'(T'(F',X)) = false \leftrightarrow X \notin \mathcal{I}[F']$.
   Since $X \in D^{\mathcal{I}}$,
   $X \notin \mathcal{I}[F'] \leftrightarrow X \in D^{\mathcal{I}} \backslash \mathcal{I}[F'] \leftrightarrow X \in \mathcal{I}[\neg F'] \leftrightarrow X \in \mathcal{I}[F]$

[3] Note that s is a one-to-one mapping and $s^{-1}(J)$ can be easily defined for any FOL interpretation J.

5. $F = \forall R.F'$:
   Assume $\forall Y \in D^{\mathcal{I}} : (\mathcal{I}'(T'(F', Y)) = true \leftrightarrow Y \in \mathcal{I}[F'])$.
   $T'(F, X) = \forall Y : (R(X, Y) \rightarrow T'(F', Y))$, so
   $\mathcal{I}'(T'(F, X)) = true$
   $\leftrightarrow \forall Y : (\mathcal{I}'(R(X, Y)) = true \rightarrow \mathcal{I}'(T'(F', Y)) = true)$
   $\leftrightarrow \forall Y : ((X, Y) \in \mathcal{I}[R] \rightarrow Y \in \mathcal{I}[F'])$
   $\leftrightarrow \forall (X, Y) \in \mathcal{I}[R] : (Y \in \mathcal{I}[F']) \leftrightarrow X \in \mathcal{I}[\forall R.F'] \leftrightarrow X \in \mathcal{I}[F]$

6. $F = \exists R.F'$:
   Assume $\forall Y \in D^{\mathcal{I}} : (\mathcal{I}'(T'(F', Y)) = true \leftrightarrow Y \in \mathcal{I}[F'])$.
   $T'(F, X) = \exists Y : (R(X, Y) \wedge T'(F', Y))$,
   so $\mathcal{I}'(T'(F, X)) = true$
   $\leftrightarrow \exists Y : (\mathcal{I}'(R(X, Y)) = true \wedge \mathcal{I}'(T'(F', Y)) = true)$
   $\leftrightarrow \exists Y : ((X, Y) \in \mathcal{I}[R] \wedge Y \in \mathcal{I}[F'])$
   $\leftrightarrow \exists (X, Y) \in \mathcal{I}[R] : (Y \in \mathcal{I}[F']) \leftrightarrow X \in \mathcal{I}[\exists R.F'] \leftrightarrow X \in \mathcal{I}[F]$

7. $F = \geq nR$:

   $T'(F, X) = \exists X_1 \ldots X_n : (R(X, X_1) \wedge \ldots \wedge R(X, X_n) \wedge (X_1 \neq X_2)$
   $\wedge (X_1 \neq X_3) \wedge \ldots \wedge (X_{n-1} \neq X_n))$,
   so $\mathcal{I}'(T'(F, X)) = true$
   $\leftrightarrow \exists X_1 \ldots X_n : (\mathcal{I}'(R(X, X_1)) = \ldots = \mathcal{I}'(R(X, X_n)) = true \wedge$
   $(X_1 \neq X_2) \wedge (X_1 \neq X_3) \wedge \ldots \wedge (X_{n-1} \neq X_n))$
   $\leftrightarrow \exists X_1 \ldots X_n : ((X, X_1) \in \mathcal{I}[R] \wedge (X, X_n) \in \mathcal{I}[R] \wedge (X_1 \neq X_2) \wedge$
   $(X_1 \neq X_3) \wedge \ldots \wedge (X_{n-1} \neq X_n))$
   $\leftrightarrow \#(\{X_i | (X, X_i) \in R\}) \geq n \leftrightarrow X \in \geq nR \leftrightarrow X \in \mathcal{I}[F]$

8. $F = \leq nR$:

   $T'(F, X) = \forall X_1 \ldots X_{n+1} : [(R(X, X_1) \wedge \ldots \wedge R(X, X_{n+1}))$
   $\rightarrow ((X_1 = X_2) \vee (X_1 = X_3) \vee \ldots \vee (X_n = X_{n+1}))]$,
   so $\mathcal{I}'(T'(F, X)) = true$
   $\leftrightarrow \forall X_1 \ldots X_{n+1} : [(\mathcal{I}'(R(X, X_1)) = \ldots = \mathcal{I}'(R(X, X_{n+1})) = true)$
   $\rightarrow ((X_1 = X_2) \vee (X_1 = X_3) \vee \ldots \vee (X_n = X_{n+1}))]$
   $\leftrightarrow \forall X_1 \ldots X_{n+1} : [((X, X_1) \in \mathcal{I}[R] \wedge (X, X_{n+1}) \in \mathcal{I}[R])$
   $\rightarrow ((X_1 = X_2) \vee (X_1 = X_3) \vee \ldots \vee (X_n = X_{n+1}))]$
   $\leftrightarrow \#(\{X_i | (X, X_i) \in R\}) \leq n \leftrightarrow X \in \leq nR \leftrightarrow X \in \mathcal{I}[F]$

□

We now proceed with the proof of the theorem.

Proof:
First, we prove the "$\leftarrow$" part. Assume $\mathcal{I}$ is a model of θ. We show that

$\mathcal{I}'$ is a model of $T(\theta)$. We start with the FOL part. We must prove that $T(c_1), \ldots, T(c_m)$ hold in $\mathcal{I}'$, where $c_1, \ldots, c_m$ are all constraints in the A-Box.

We know $\mathcal{I}$ is a model of θ, so for all $F \in abox(\theta) \; : \; \mathcal{I} \models F$. This means that:

1. if $a:F \in abox(\theta)$ then $a \in \mathcal{I}[F]$
2. if $aRb \in abox(\theta)$ then $(a,b) \in \mathcal{I}[R]$

Now, consider any FOL axiom $T(c_i)$. If $T(c_i) = C(a)$ then $c_i = a : C \in abox(\theta)$. Therefore, $a \in \mathcal{I}[C]$ and by definition $\mathcal{I}'(C(a)) = true$. Similarly, if $T(c_i) = R(a,b)$, then $c_i = aRb \in abox(\theta)$. Therefore, $(a,b) \in \mathcal{I}[R]$ and by definition $\mathcal{I}'(R(a,b)) = true$.

It remains to be proven that $\mathcal{I}'$ is a model of $T(tbox(\theta))$: for each defined predicate the completion of its definition must hold in $\mathcal{I}'$.

Since every concept definition has been mapped to one general clause and no other clauses were generated, each clause $C(X) \leftarrow T'(F,X)$ in $T(tbox(\theta))$ corresponds to one definition $C == F$ in $tbox(\theta)$. We need to prove that for each such clause, $\mathcal{I}'(C(X)) = true \leftrightarrow \mathcal{I}'(T'(F,X)) = true$. We know that $\mathcal{I}'(C(X)) = true \leftrightarrow X \in \mathcal{I}[C]$, and since $C == F$ is in $tbox(\theta)$, $X \in \mathcal{I}[C] \leftrightarrow X \in \mathcal{I}[F]$. Applying lemma 1, we obtain the desired result.

We then prove the "$\rightarrow$" part of the equivalence. Assume $\mathcal{I}'$ is a model of $T(\theta)$. We show that there exists an interpretation $\mathcal{I}$ of $tbox(\theta)$ such that $s(\mathcal{I}) = \mathcal{I}'$, and which is a model of θ.

We can obtain $\mathcal{I}$ as follows. For every concept symbol C, for every role symbol R and for every $X, Y \in D^{\mathcal{I}}$, we know that $\mathcal{I}$ should satisfy:

$$X \in \mathcal{I}[C] \leftrightarrow \mathcal{I}'(C(X)) = true$$
$$(X,Y) \in \mathcal{I}[R] \leftrightarrow \mathcal{I}'(R(X,Y)) = true$$

Then we apply the definitions of interpretations to extend $\mathcal{I}$ to general concept descriptions. We prove that $\mathcal{I}$ then satisfies the definitions $C == F$ in $tbox(\theta)$ and that $\mathcal{I} \models F$, for all $F \in abox(\theta)$.

Given that $\mathcal{I}'$ is a model of $T(\theta)$, we know that $T(c_1), \ldots, T(c_m)$ hold in $\mathcal{I}'$, where $c_1, \ldots, c_m$ are all constraints in the A-Box. Any $F \in abox(\theta)$ is of one of the forms $a : C$ or aRb. If $F = (a : C)$, we can use the knowledge $\mathcal{I}'(T(a:C)) = \mathcal{I}'(C(a)) = true$, from which we obtain using the definition that $a \in \mathcal{I}[C]$. If $F = (aRb)$, we use the knowledge $\mathcal{I}'(T(aRb)) = \mathcal{I}'(R(a,b)) = true$, from which we obtain that $(a,b) \in \mathcal{I}[R]$. In both cases it follows that $\mathcal{I} \models F$.

It remains to be proven that $\mathcal{I}$ satisfies all definitions $C == F$ in $tbox(\theta)$. For each such definition, $T(tbox(\theta))$ contains $C(X) \leftarrow T'(F, X)$, so since $\mathcal{I}$ is a model, $\mathcal{I}'(C(X)) = true \leftrightarrow \mathcal{I}'(T'(F, X)) = true$. Using the definition of $\mathcal{I}$ and lemma 1, we find respectively $\mathcal{I}'(C(X)) = true \leftrightarrow X \in \mathcal{I}[C]$ and $\mathcal{I}'(T'(F, X)) = true \leftrightarrow X \in \mathcal{I}[F]$. From these three equivalences it follows that $X \in \mathcal{I}[C] \leftrightarrow X \in \mathcal{I}[F]$.
□

It is important to observe that in DLs no domain closure assumption is present: it is assumed there may be unknown objects that are not mentioned in the theory. This is also possible in OLP, since OLP allows for non-Herbrand interpretations (see [26]). Thus representing open domains is possible, like in description logics.

At this point it should also be mentioned that sometimes the T-Box in description logics is allowed to contain other formulae than definitions, in particular formulae of the form $C \sqsubseteq D$, imposing as a constraint that the concept C is a subset of D (for example $father \sqsubseteq parent$). Such formulae are general statements about the domain and not assertions about specific objects. Therefore they belong in the T-Box. However their equivalent counterpart in OLP are FOL axioms of the form $\forall X : (C(X) \rightarrow D(X))$.[4] This is an exception to the general principle that the T-Box is mapped to program clauses and the A-Box to FOL axioms. If these constructs are allowed, the T-Box corresponds to program clauses plus universally quantified FOL axioms, whereas the A-Box corresponds to ground FOL atoms.

3.4.2 Description Logics as Sublanguages of OLP

The set of open logic programs obtained from DL theories using a mapping like the above one is only a small subset of the set of all possible logic programs. In particular, an open logic program corresponding to an $\mathcal{ALCN}$ theory satisfies the following conditions (possibly after a permutation of literals in the body of clauses):

- The program contains only unary and binary literals and inequalities, and there are no functors of arity > 0.
- The head of each clause is a unary atom.
- The body of each clause is a sequence of restricted literals of that clause. A restricted literal of a clause Cl is one of the following:

[4] The proof is straightforward.

- A binary atom A of which the first parameter is equal to either the parameter of the head of Cl or the parameter of a unary atom preceding A in Cl, and of which the second parameter does not occur in any literal preceding A in Cl.
- A positive or negative unary literal L of which the parameter is equal to either the parameter of the head of Cl, or to the second parameter of a binary literal preceding L in Cl.
- An inequality of which both parameters occur as the second parameter of a binary literal preceding L in Cl, and do not occur anywhere else in Cl.

- No predicate depends on itself.
- All FOL axioms are either
  - ground binary or unary atoms, or
  - formulae of the form $\forall X : (C(X) \rightarrow D(X))$ with C and D unary predicates

For other description logics, similar corresponding sublanguages of OLP can be determined. For example, the language $\mathcal{ALC}$ ($\mathcal{ALCN}$ without number restrictions) corresponds to the language described above without inequalities. In $\mathcal{ALCQ}$ ($\mathcal{ALC}$ with qualified number restrictions), the parameters of an inequality in Cl may also occur in unary literals in the body of Cl provided they both occur as parameter of exactly the same set of predicates, and both with the same sign(s) for each such predicate.

Role constructors extend the corresponding sublanguage in different ways: role concatenation introduces binary atoms of which the first parameter can also be the second parameter of a preceding binary atom in Cl. Inverse roles result in binary atoms in which the conditions on the first parameter are satisfied by the second parameter and vice versa (conditions on literals further in Cl then also refer to the first parameter of that atom rather than the second). We do not go into any more details here. Note that the introduction of the role constructors union, intersection and concatenation does not enlarge the set of concepts that can be defined: these constructors merely allow one to write shorter definitions, using less auxiliary concepts.

Having determined the OLP-sublanguages corresponding to several DLs, we can also define inverse mappings from theses sublanguages into DL theories. These mappings are rather straightforward: each predicate definition can be mapped to a concept description independently. As an example,

the sublanguage corresponding to $\mathcal{ALCN}$ defined above can be mapped to $\mathcal{ALCN}$ as follows.

A set of clauses $\{C(X) \leftarrow F_1, \ldots, C(X) \leftarrow F_t\}$ is mapped to the concept definition $C == M(X, F_1) \sqcup \ldots \sqcup M(X, F_t)$, where $M(X, F)$ is defined inductively as:
$M(X, F) = M'(C_1, F) \sqcap \ldots \sqcap M'(C_n, F) \sqcap M'(D_1, F) \sqcap \ldots \sqcap M'(D_m, F)$ where $C_1 \ldots C_n$ are the unary literals in F containing X as parameter, and given that $R_1 \ldots R_m$ are the binary predicates occurring in F with X as first parameter, each D_i, $i = 1 \ldots m$, is the conjunction of all literals of the form $R_i(X, Z)$ (with Z a variable) in F. If $n = m = 0$ then $M(X, F) = \top$. M' is itself defined as

- $M'(C(X), F) = C$.
- $M'(\neg C, F) = \neg M'(C, F)$.
- $M'((R(X, Y_1) \wedge \ldots \wedge R(X, Y_k)), F) =$
  - $\exists R.(M(Y_1, F))$ if $k = 1$.
  - $\geq nR$ if $k = n \neq 1$ and for all $1 \leq i, j \leq n$: $Y_i \neq Y_j$ or $Y_j \neq Y_i$ occur in F.
  - Otherwise, $M'((R(X, Y_1) \wedge \ldots \wedge R(X, Y_i)), F) \sqcap M'((R(X, Y_{i+1}) \wedge \ldots \wedge R(X, Y_k)), F)$, where i is such that no inequality $Y_r \neq Y_s$ with $r \leq i < s$ or $s \leq i < r$ is in F.

It can easily be checked that $\mathcal{ALCN}$ definitions obtained from OLP clauses using the mapping M yield the original OLP clauses again when the mapping T is applied to them: $T(M(theory)) = theory$. The reverse, i.e. that $M(T(theory) = theory$ for any $\mathcal{ALCN}$-theory, is not true since M maps clauses to only a particular subset of $\mathcal{ALCN}$. However $M(T(theory)$ is an $\mathcal{ALCN}$ theory equivalent to $theory$.

The mapping can easily be extended to theories with FOL axioms. For ground unary and binary atoms we obtain: $M(C(a)) = a : C$ and $M(R(a, b)) = aRb$. For the other FOL axioms we obtain $M(\forall X : (C(X) \rightarrow D(X)) = C \sqsubseteq D$. In these cases, M is exactly the inverse of T.

3.5 Comparison of SLDNFA to a DL Algorithm

The correspondence between DLs and OLP is also visible at the procedural level. We describe the correspondence between SLDNFA and the procedure for consistency checking of $\mathcal{ALCN}$ theories described in [48]. This

algorithm does not consider the presence of a (non-empty) A-Box, but it can be extended rather easily to deal with one. The same holds for the corresponding selection rules we will introduce in the SLDNFA-procedure.

Most tasks studied in DLs can be reduced to checking if a certain concept C is satisfiable (i.e. if there can exist elements satisfying the definition of C). In a logic-based framework this corresponds to checking if $\exists X : C(X)$ is consistent with the theory.

In $\mathcal{ALCN}$, this task is solved in three steps. First, the definition of C is completely unfolded, i.e. rewritten using the definitions as rewriting rules (this process always terminates since no recursion is allowed). Then, a first constraint $x : F$ is generated, where x is a variable and F the unfolded definition of C. Finally, an algorithm using constraint propagation rules is used to derive new constraints from the existing ones, until no more propagation rules apply or a contradiction is found. The propagation rules and the algorithm depend on which description logic is used.

We briefly describe the algorithm for $\mathcal{ALCN}$ as it is presented in [48]. First we define the following concepts: assume S is a constraint system, R is a role, and x a variable occurring in S. The number of constraints in S of the form $x : \exists R.C$ is defined as

Definition 3.5.1

$$ex_S^R(x) = \#(\{C | x : \exists R.C \in S\})$$

Another concept we need is the minimal "atmost" constraint imposed on x by R in S.

Definition 3.5.2 *Define* $N = \{n | x : (\leq nR) \in S\}$. *Then*

$$atmost_S^R(x) = if\ N \neq \{\}\ then\ min\ (N)\ else\ \infty$$

Definition 3.5.3 (clash) *A constraint system S contains a* clash *if it contains a subset of one of the forms*

- $\{x : C,\ x : \neg C\}$
- $\{x : \exists R.C,\ x : \leq 0R\}$
- $\{x : \geq mR,\ x : \leq nR\}$ *where* $m > n$

To check the satisfiability of a concept, a functional algorithm is used: C is satisfiable if $sat(x, \{x : F\})$ holds, where F is the unfolded definition of C.

$sat(x, S) =$
  if S contains a clash then $false$

elsif $x : C \sqcap D \in S$ and $x : C \notin S$ or $x : D \notin S$
then $sat(x, S \cup \{x : C, x : D\})$
elsif $x : C \sqcup D \in S$ and $x : C \notin S$ and $x : D \notin S$
then $sat(x, S \cup \{x : C\})$ or $sat(x, S \cup \{x : D\})$
else (for a new introduced variable y)
$\forall (x : (\geq nR)) \in S$ with $ex_S^R(x) = 0$:
$sat(y, S \cup \{y : C | x : \forall R.C \in S\})$
and $\forall (x : \exists R.C) \in S$ with $ex_S^R(x) \leq atmost_S^R(x)$:
$sat(y, S \cup \{y : C\} \cup \{y : D | x : \forall R.D \in S\})$
and $\forall \{x : \exists R.C_1, \ldots x : \exists R.C_l\} \subseteq S$
with $l = ex_S^R(x)$, $k = atmost_S^R(x), l > k$:
$\exists$ a k-partition $\prod$ of $\{C_1, \ldots C_l\}$ such that
$\forall \pi \in \prod : sat(y, S \cup \{y : C | C \in \pi\} \cup \{y : D | x : \forall R.D \in S\})$

where a *k-partition* of X is a set $\prod$ containing k pairwise disjoint subsets π of X such that $\bigcup_{\pi \in \prod} \pi = X$.

We illustrate the use of this algorithm on an example. Assuming the T-Box of our previous example, i.e.

$$\begin{array}{lcl} C & == & (\exists child.D) \sqcap (\forall child.E) \\ D & == & male \sqcup \geq 3\ child \\ E & == & \leq 1\ child \end{array}$$

we check the consistency of concept C. To this end we first unfold its definition :

$$C \ == \ (\exists child.[male \sqcup \geq 3\ child]) \sqcap (\forall child.[\leq 1\ child])$$

We apply the algorithm to constraint system S_0, with working variable x:

$$S_0 = \{x : (\exists child.[male \sqcup \ \geq 3\ child]) \sqcap (\forall child.[\leq 1\ child])\}$$

$$S_1 = \ S_0 \cup \{x : (\exists child.[male \sqcup \ \geq 3\ child]), x : (\forall child.[\leq 1\ child])\}$$

Then the algorithm introduces a new working variable y:

$$S_2 = \ S_1 \cup \{y : [male \sqcup \ \geq 3\ child]\} \cup \{y : [\leq 1\ child]\}$$

Note that x *child* y is not explicit in the constraint system. This formula will be true in any interpretation satisfying C. However, it is left implicit as it has no further influence on the algorithm.

Now, a "$\sqcup$" occurs in the constraint system, and one of two branches in the search tree has to be chosen. We explore both branches consecutively:

$$S_{3a} = S_2 \cup \{y : [\geq 3\ child]\}$$

At this point the system contains the subset

$$\{y : [\leq 1\ child], y : [\geq 3\ child]\}$$

which is a clash. This constraint system is therefore unsatisfiable.

The other branch yields:

$$S_{3b} = S_2 \cup \{y : male\}$$

At this point, no more propagation rules apply and a solution is found. $x : C$ holds if $x\ child\ y$ and $y : male$ hold, which does not violate any other constraint (*male* is a primitive concept). Hence C is satisfiable.

Let us now check how SLDNFA handles this example. We start from the open logic program obtained before as the translation of the above theory, i.e.

$$\begin{array}{lcl} C(X) & \leftarrow & child(X,Y), D(Y), \neg\, H(X). \\ H(X) & \leftarrow & child(X,Z), \neg\, E(Z)). \\ D(X) & \leftarrow & male(X). \\ D(X) & \leftarrow & child(X,Y), child(X,Z), child(X,U), Y \neq Z, Y \neq U, Z \neq U. \\ E(X) & \leftarrow & \neg\, G(X). \\ G(X) & \leftarrow & child(X,Y), child(X,Z), Y \neq Z. \end{array}$$

and check if $\exists X : C(X)$ is consistent with this theory. The open predicates are the primitive ones of the DL theory: *child*/2 and *male*/1. As is done with the DL definition, we unfold the definition of $C(X)$ first:[5]

$$\begin{array}{lcl} C(X) & \leftarrow & child(X,Y), [male(Y) \vee \\ & & child(Y,T), child(Y,U), child(Y,V), T \neq U, U \neq V, T \neq V], \\ & & \neg \exists Z, A, B : [child(X,Z), child(Z,A), child(Z,B), A \neq B]. \end{array}$$

We then have to find a derivation for $\leftarrow \exists X : C(X)$.

Since we start with a completely unfolded definition, all defined predicates have been replaced and all remaining literals are instances of open predicates or inequalities. We recall the most important features of SLDNFA, adapted to this simplified setting. For more details we refer to Chapter 2. The procedure maintains three sets: PG of positive goals (goals which

[5] In practice the unfolding of goals is delayed until they are selected, with little influence on the derivation. For our discussion it is more appropriate to work with the unfolded definition, as it highlights the correspondence with the $\mathcal{ALCN}$ algorithm more directly. Note that the unfolded definition never contains universal quantifiers, and that existential quantifiers can be left implicit as usual in a logic program. For clarity reasons we write existential quantifiers explicitly as long as they occur in a negated formula.

need to succeed), NG of negative goals (which need to fail), and Δ of abduced atoms. As all literals in the unfolded definition are either instances of open predicates or inequalities, all with only variables as arguments, all goals in our derivation consist only of open literals and of equalities and inequalities with variables and/or skolem constants as arguments. Moreover equalities only occur in NG (equalities in PG are eliminated by unifying their arguments and applying the unifying substitution to PG, NG and Δ). Observe moreover that different variables and/or skolem constants can always consistently be assumed to be different, so any equality atom not of the form $X = X$ can be assumed to be false.

As a result, each negative goal can be assumed to fail unless all of its atomic conjuncts unify with abduced atoms or are equalities. Evidently, the empty goal always succeeds. Initially Δ is empty, so each negative goal containing an atom can be assumed to fail. Each time a newly abduced atom unifies with an atom in a negative goal, we apply resolution to obtain an additional negative goal. If this goal is empty, the derivation fails. If we can guarantee that negative goals are always generated before any abduced atoms they may unify with, it suffices to check the negative goals when new atoms are abduced (apart from checking if no equality is of the form $X = X$). Then we know that at any time, given the current Δ, each negative goal containing an atom can consistently fail. Therefore, whenever PG is empty and all negative goals contain an atom, a solution is found. Failure is obtained as soon as a negative goal is empty.

We start with the positive goal $\leftarrow C(X)$ and no negative goals nor abduced atoms. Using the definition of $C(X)$ we obtain

$$\begin{array}{ll} PG_0 = & \{\leftarrow child(X,Y), [male(Y) \vee \\ & child(Y,T), child(Y,U), child(Y,V), T \neq U, U \neq V, T \neq V], \\ & \neg\exists Z, A, B : [child(X,Z), child(Z,A), child(Z,B), A \neq B]\} \\ NG_0 = & \Delta_0 = \quad \{\} \end{array}$$

A part of the SLDNFA-derivation tree for this goal is given in Figure 3.1.[6]

The figure shows that the derivations by the DL algorithm and SLDNFA in this example are nearly identical. The only exception is the way number restrictions are handled. The DL algorithm detects a clash in the number restrictions (node 3a in the derivation tree) immediately in the syntax. SLDNFA does not contain a special treatment of number restrictions and has to expand the corresponding constraints to find a contradiction. This involves several abduction steps and the generation of lots of negative goals,

[6]In the figure, the formula $has_3_child(Y)$ is used as a shorthand notation for $child(Y,T), child(Y,U), child(Y,V), T \neq U, U \neq V, T \neq V$. $has_2_child(Y)$ is used in a similar way.

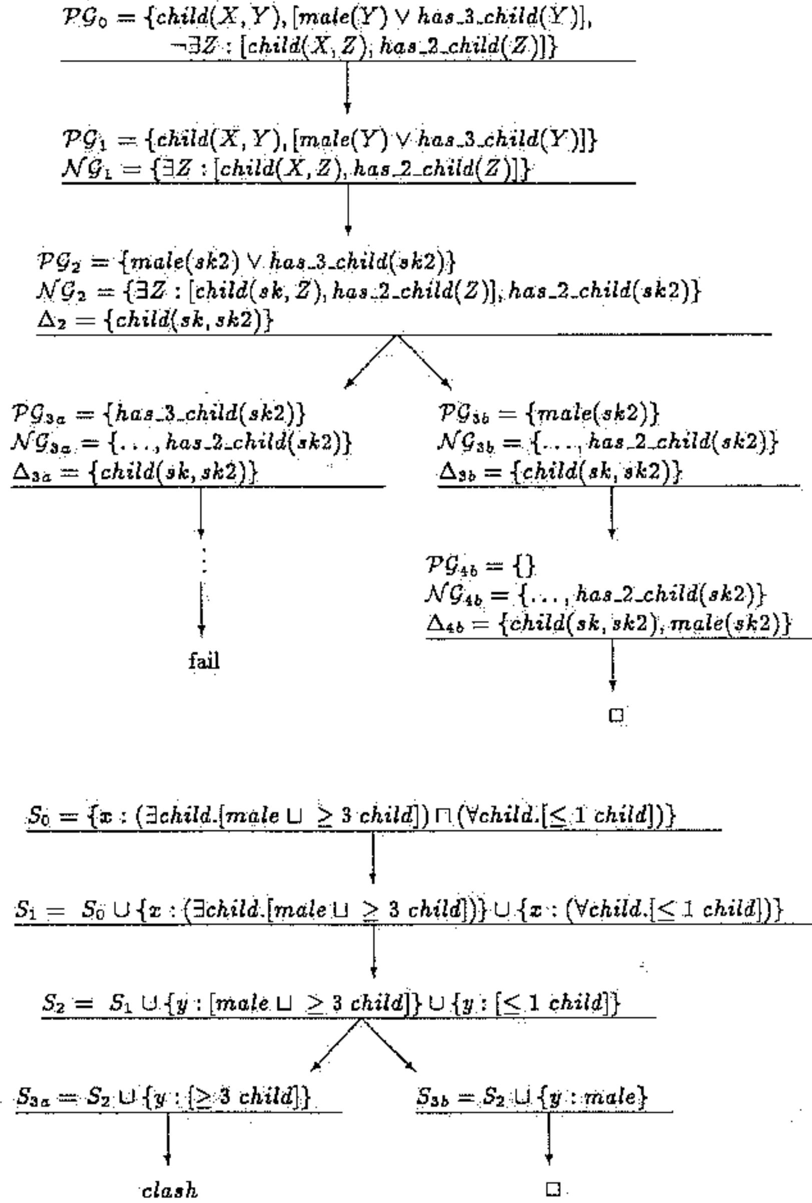

Figure 3.1: Derivation trees for SLDNFA and the $\mathcal{ALCN}$ algorithm

which leads to a considerable efficiency loss. Except for this, there is a clear node-per-node correspondence between the derivation trees.

This result can be generalised. A control strategy equivalent to the one in $\mathcal{ALCN}$ can be obtained by SLDNFA (working on the corresponding sub-language of OLP) by using an appropriate selection rule. More specifically, the following algorithm describes such a rule:[7]

1. Choose a variable X not occurring as second parameter in any literal.
2. For each disjunction in which X occurs, consecutively select each disjunct (deleting the other ones), and for the resulting goal do:
   - While X occurs in any formula $\neg F$ in PG, select one such formula. ($\leftarrow \neg F$ is deleted from PG and $\leftarrow F$ is added to NG.)
   - While X occurs in any literal in PG, select one such literal. (The corresponding fact is abduced; if it unifies with a literal of a goal in NG, negative resolution is applied yielding an additional goal in NG).
   - While X occurs in an inequality $X \neq Y$ in NG, select one such goal. (X and Y are unified — this leads to failure if $\leftarrow X = Y$ is in NG).
   - While X occurs in a goal $X \neq Y$ in PG, select one such goal. (If X and Y are the same variable, the derivation fails, otherwise $\leftarrow X = Y$ is added to NG).
   - Delete all literals containing X and goto 1.

The $\mathcal{ALCN}$ algorithm above as well as our proposed selection rule assume an empty A-Box. The extension to problems with a non-empty A-Box is straightforward: to check the consistency of a theory in $\mathcal{ALCN}$ the A-Box is the initial constraint set and constants are treated like variables. In SLDNFA the conjunction of unfolded FOL axioms (or, if the FOL axioms are written in the form of a definition for *invalid*, the unfolded definition of $\neg$*invalid*) is the initial goal, and in the selection rule constants are treated as separate variables.

3.6 Discussion

In this chapter we have shown a declarative equivalence between DL theories and open logic programs. A mapping from $\mathcal{ALCN}$ theories into open

[7]Equality atoms are never selected: as we discussed above they only appear in negative goals and can then be assumed to be false.

logic programs is defined. For a number of DLs we have identified equivalent sublanguages of OLP. A mapping to the corresponding DL is possible for these sublanguages, as illustrated for one language. We have also studied the correspondence between DL algorithms and SLDNFA. Using an appropriate selection rule in SLDNFA we obtain derivations in which consecutive intermediate sets of goals correspond to consecutive constraint sets of a DL algorithm.

A strength of DLs is the use of efficient (optimal) algorithms for each specific language. Detailed complexity results for most languages are available, allowing to choose a language for a particular application based on both complexity and expressiveness considerations. The inefficiency of (open) logic programming is for a large part due to its greater expressive power. As we have shown, for a particular sublanguage of OLP an appropriate selection rule leads to very similar derivations and hence similar complexity results to those obtained in the equivalent description logic. One noteworthy exception is formed by the number restrictions. On the declarative level, the OLP equivalent of a number restriction is a lot less compact, which is a disadvantage for knowledge representation. On the procedural level, handling number restrictions is also substantially slower in OLP.

The correspondences clarify long-questioned relations and open up possibilities for cross-fertilisation between these two very related yet up to this day completely separate areas of research. One interesting result could be the extension of OLP with a representation of number restrictions, and of SLDNFA with an efficient way of handling them. In fact an obvious way to achieve this is by incorporating CLP techniques in open/abductive logic programming, as in [51]. This is one major issue for further research.

A possible gain for the DL community is that the relation we point out shows directions for further upgrading the knowledge representation power of current languages, bringing them closer to the richer OLP formalism. In fact we notice that recently added constructs to DLs, like reflexive transitive closure of roles, map to more expressive subsets of OLP for which stronger LP-semantics than the completion (for example the justification semantics or the generalised stable model semantics of [49]) are appropriate.

The correspondence to DLs also shows that OLP is a highly expressive knowledge representation language. Like DLs, it distinguishes between an A-Box and a T-Box component of the represented knowledge. Also like DLs it allows for reasoning on open domains and with incomplete knowledge, two important issues in knowledge representation. To us this is the most interesting result, as the use of OLP for knowledge representation is the central subject of this thesis.

Chapter 4

Time in Knowledge Representation

The previous chapter discussed the suitability of open logic programming as a general knowledge representation language by comparing it with existing KR languages, and by showing how it satisfies the important knowledge representation principle of dealing with both terminological and assertional information. In the rest of this thesis, we complement this theoretical suitability result by applying open logic programming to practical as well as open theoretical problems in knowledge representation.

Knowledge representation in practice is a very domain dependent task. However, there also exist a number of open questions which are relevant to very large classes of applications. One of the most important of these questions, and apparently one of the toughest, is how to deal with time.

4.1 Outline of the Problem

Dealing with time is necessary in all problem domains which are in some way dynamic, i.e. subject to change. Roughly speaking, this is the case for practically all (interesting) real world domains. In particular the agent itself, which is working and solving problems in the domain, is a major cause of change. This makes a correct representation of dynamic problem domains an issue of considerable importance.

What makes dynamic domains special is that one needs to distinguish between the state of the domain at different time points. There are strong relations between these states at different times, mainly due to the law of inertia. In general, we expect that anything which is not caused to

change at a certain time will remain in the same state. An agent should be aware of this, but it should not be the task of a user representing a specific problem domain to exhaustively enumerate everything which remains inert at each time: a good knowledge representation formalism should allow for correctly deriving what changes and what remains inert, given concise and intuitive rules for change provided by the user. Since the late sixties, a lot of research in the AI community is devoted to tackling these problems in temporal reasoning: to find a correct formalisation of the general principles underlying time and change.

McCarthy and Hayes have identified the central problem, called the *frame problem*, in [72]. They distinguish three sub-problems, specified below:

- The inertia problem is the problem of automatically determining which parts of the world do not change as a result of a particular action or sequence of actions of which the effects in each particular context are known. For example, if we know that a block A is in a certain location, we expect that moving a different block B will not change the location of A. An agent should make the same assumption. To this end its theory should contain an appropriate formalisation of the closed world assumption.

- The qualification problem is defined as the problem of determining the effects of an action, given a not necessarily completely known list of exceptional circumstances, which in the absence of explicit information are assumed not to occur. An often given example is that we can normally start our car, but under exceptional circumstances, for example if someone has put a potato in the tailpipe, we cannot get the car going. We assume in general that there is no potato in the tailpipe, unless we have good reason to believe there might be. We do not explicitly check for potatoes (or who knows what other exceptional situations) any time we start the car, but we assume that the car will start when we turn the key. Again, an agent should be able to make the same assumptions by default.

- The ramification problem is the problem of determining *all* effects of a particular action: in other words, also possible indirect effects which automatically arise as a result of specified direct effects. These effects form exceptions to the inertia assumption. A simple example is the following: assume someone (or, to make it more acceptable, a turkey) gets shot, and the immediate effect is that it dies. If at the time it got shot the turkey was walking around, we expect that since it dies, it

will also stop walking. An agent should be able to determine exactly the expected effects.

The inertia problem is in general tackled by applying some form of minimisation of change representing the closed world assumption. The problem is to find a suitable minimisation schema, which has proven to be a nontrivial task: it was for example shown in [43] and [44] that all the approaches developed up to that time, failed (i.e. did not entail the intended conclusions) on a number of very small examples, like the since then famous Yale Shooting Problem (of which the above example of a walking turkey is an extension). The qualification problem as defined in [72] is mostly a problem of default reasoning, and in that sense not really typical for reasoning about time. The term "qualification problem" in temporal reasoning has often been assigned a different meaning, namely the problem of determining under what circumstances an action can or cannot occur given particular constraints on the state of the world. The ramification problem, which is currently assigned the most attention, adds a lot of complexity to the minimisation policy needed to deal with inertia, since indirect effects form complex exceptions to the general inertia principle.

In the area of temporal reasoning, a lot of other problematic issues have come up, some of which further complicate the frame problem and some of which are orthogonal to it. These issues comprise dealing with simultaneous actions, with nondeterminism, with incomplete information on action occurrences, with an unknown initial state of the world, with delayed effects of actions and with continuous change. Apart from that there are also more fundamental issues, like what the topology of time itself should be (for example linear or branching, continuous or discrete).

Several formalisms and a lot of variants of these formalisms have been proposed to deal with some or all of the problems mentioned above. In any approach, two main choices need to be made: a choice of the basic ontology of time and actions, and a choice of minimisation policy to deal with the frame problem. The approaches tend to be named after their basic ontology, and variants are distinguished by the choice of minimisation policy and possibly minor ontological differences.

By far the most widely used ontology is that of the Situation Calculus, introduced in [72]. It has been used as the basis of many different attempts at solving the frame problem, to give but some examples in [31], [4], [98], [6], [83], [90], [65]. Minimisation policies used in Situation Calculus are often variants of circumscription ([71]), which use a second-order logic axiom. Alternatively, implicit or explicit predicate completion and sometimes stronger techniques based on logic programming semantics are used. We describe and discuss the Situation Calculus in detail in the following chapter,

where we will indicate similarities and differences with our own formalism, a variant of the Event Calculus. The Event Calculus is another widely used formalism, introduced in [59], which we will use throughout the rest of the thesis. In Event Calculus, the closed world assumption is usually imposed by some logic programming semantics, though circumscription is also applied in some approaches, in particular in [102], [103] and [74]. We give more details on the Event Calculus below. Other important formalisms, to which we will refer further on, are Allen's interval-based theory of time ([3]) and Sandewall's *Features and Fluents* approach ([96]).

For more details on the history of the frame problem and the many approaches proposed to deal with it, we refer to [103], in which both successful approaches (coming closer to a complete solution) and failed ones and their lessons are discussed. The book also offers a new approach based like the work in this thesis on the Event Calculus, but not in an open logic programming setting. We have not yet studied the relation between this recent approach and our work.

4.2 The Event Calculus

The formalism we choose to work with in this thesis is an open logic programming variant of the Event Calculus. The Event Calculus was originally defined by Kowalski and Sergot in [59]. Central to the formalism is the notion of action occurrences, or *events*, at certain points on a time line. Events are assumed to be instantaneous, i.e. like in most approaches abstraction is made of the possible duration of actions. Events determine time intervals during which certain *fluents* (time dependent statements about the state of the world) hold. In the original Event Calculus, these time intervals are explicitly represented, though in later simplified versions, like the one we use, they are only implicit. Since the original definition a lot of variants have been used, for example by Shanahan in [99, 100, 102, 103], by Miller e.a. in [73, 50], by Montanari e.a. in [80], by Evans in [32], by Missiaen e.a. in [77, 79] and by Kowalski e.a. in [58, 57, 93].

4.2.1 Formalisation

The Event Calculus we propose is close to the one used in [28], formalised as an open logic program. We first describe the basic concepts and the predicates representing them. First of all we have a type predicate $time/1$: $time(T)$ means T is a time point. A predicate $holds/2$ determines the truth of fluents at certain times: $holds(P,T)$ represents that fluent P holds at time T. An event is the occurrence of an action at a certain point in time.

The occurrence of an event E at time T is denoted by $happens(E, T)$. The atom $act(E, A)$ denotes that E consists of the occurrence of an action of type A. An event can *initiate* or *terminate* existing fluents, i.e. cause them to be true or false, depending on the action associated with it. $initiates(E, P)$ ($terminates(E, P)$) means that event E initiates (terminates) the fluent P.

The central axiom of the Event Calculus, the so-called frame axiom, which captures the law of inertia, is the following pair of clauses:

$$\begin{array}{lll} holds(P,T) & \leftarrow & happens(E_1,T_1), T_1 < T, initiates(E_1,P), \\ & & \neg clipped(T_1,P,T). \\ clipped(T_1,P,T) & \leftarrow & happens(E_2,T_2), T_1 < T_2, T_2 < T, \\ & & terminates(E_2,P). \end{array}$$

i.e. a fluent holds at a certain time point T if (and only if) it has been initiated by an earlier event and if it has not been terminated between that initiation and T.[1]

In addition, the Event Calculus contains a number of axioms determining the topology of time, and some general constraints on problem domains. Time is considered to be one (usually infinite) line, hence the order on time points must be a linear order. We impose this condition despite the fact that in some formalisations of Event Calculus only a partial order on time points is imposed and argued to be sufficient. The argument is usually that in many cases the ordering of two particular time points is not relevant, so that we should not impose any one order. However, the restriction to a linear order does not imply that a particular order is imposed: it only requires that in each model every pair of different time points is ordered in one way or the other; evidently the exact order can vary between different models. In other words, the argument confuses ignorance of the order with its non-existence. In classical logic programming, ignorance cannot be represented, and imposing a partial order is the best possible approximation. However, as it is not possible that a real time point is neither before nor after another one, the formalisation is not entirely correct. Hence it is not surprising that, as shown in [28], it is easy to devise examples where the use of a partial order leads to erroneous conclusions. In open logic programming, where incomplete knowledge can be correctly represented, these problems are avoided.

The linear order condition is represented by the following FOL axioms (recall that formulae with free variables denote their universal closure in

[1] Observe that in the given formalisation the change in truth value determined by an initiation or termination is visible immediately after the event causing it: at the time of the event itself, the old value is still valid.

what follows) :

$$\neg((T_1 < T_2) \wedge (T_2 < T_1))$$
$$((T_1 < T_2) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_3)$$
$$(time(T_1) \wedge time(T_2)) \rightarrow [(T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)]$$
$$(T_1 < T_2) \rightarrow (time(T_1) \ \wedge \ time(T_2))$$

The last axiom indicates that $<$ can only relate time points.

Another evident axiom is that each event can only occur once:

$$(happens(E,T) \ \wedge \ happens(E,T^*)) \rightarrow T = T^*$$

In some applications we will also impose that there can only be one event at each time point, in which case the previous axiom is replaced with

$$(happens(E,T) \ \wedge \ happens(E^*,T^*)) \rightarrow (E = E^* \leftrightarrow T = T^*)$$

However, this is not strictly required in general. Another condition is one of consistency: a fluent should never be initiated and terminated at the same time, which we impose by

$$\neg(initiates(E,P) \wedge terminates(E,P))$$

Usually we assume there is a given state to begin with. A predicate $initially/1$ is used to determine this initial state ($initially(P)$ denotes that P is initially true). In practice, we say that at the beginning of time, say at t_0, there is a *start* event which initiates all fluents that are initially true. This is represented by the following FOL axioms:

$$time(t_0)$$
$$happens(start, t_0)$$
$$initiates(start, P) \leftarrow initially(P)$$

No events are allowed to occur before t_0:

$$happens(E,T) \rightarrow ((t_0 < T) \vee (E = start))$$

Note that the initial event can be omitted, creating a time line which is unbounded in past and future. In most applications however, an initial state of the world comes in handy. Moreover intuitively one can also interpret this initial state to be the first state we are interested in: what went before is of no importance.

Apart from these general axioms, an Event Calculus theory should contain a description of a particular problem domain. This description consists

of two parts. A first part describes the general laws ruling the domain: most importantly this part contains a set of clauses defining the predicates *initiates* and *terminates* in terms of occurring actions and possibly of the current state of the world. Apart from this, it can also contain FOL axioms describing *action preconditions*, i.e. conditions which need to be satisfied for certain actions to be allowed; and general constraints on the domain, called *state constraints*, describing fixed relations between fluents.

As an example, take the walking turkey problem we mentioned before. Fluents in this domain are *alive* (the turkey lives), *walking* (the turkey is walking) and *loaded* (the gun is loaded). Actions are *load* (loading the gun), *shoot* (shooting the turkey), *wait* (doing nothing) and *go* (an action of the turkey: it starts walking). For simplicity reasons we assume that there are no simultaneous actions:

$$(act(E, A) \wedge act(E, A^*)) \rightarrow A = A^*$$

The direct effects of actions are described as follows:

$$\begin{array}{lcl} initiates(E, loaded) & \leftarrow & act(E, load). \\ terminates(E, alive) & \leftarrow & act(E, shoot), happens(E, T), \\ & & holds(loaded, T). \\ initiates(E, walking) & \leftarrow & act(E, go). \end{array}$$

Note that (evidently) waiting has no effect at all. A state constraint is that the turkey can only be walking if it is alive:

$$holds(walking, T) \rightarrow holds(alive, T)$$

and an action precondition would be

$$happens(E, T) \wedge act(E, go) \rightarrow holds(alive, T)$$

i.e. only living turkeys start walking. Finally, there is an indirect effect, described by the following extra clause for terminates:

$$terminates(E, walking) \leftarrow terminates(E, alive).$$

In this case, both the action precondition and the derived effect are related to the state constraint. The precondition is in fact entailed by the state constraint given the definitions of *initiates* and *terminates*. We return to this issue in a lot more detail in Chapter 7, where we tackle the ramification problem.

The second part of the application-specific theory contains scenario information: data about specific observations in the domain in a certain period of time. These data are typically a (complete or partial) specification

of the initial state, assertions of observed action occurrences, and possibly observations of fluent values at particular times. As an example, we can define the initial state as

$$\begin{array}{l} initially(walking). \\ initially(alive). \end{array}$$

which says that *walking* and *alive* are true in the beginning, but *loaded* is false (by the closed world assumption). Further information could be that the gun is at some time loaded and at a later time fired:

$$\begin{array}{lll} time(t_1) & time(t_2) & t_1 < t_2 \\ happens(e_1, t_1) & act(e_1, load) & \\ happens(e_2, t_2) & act(e_2, shoot) & \end{array}$$

In this case we do not know if there are other actions before, after, or between the two specified ones. Alternatively, we could make *happens* and *act* defined predicates, in which case

$$\begin{array}{lll} time(t_1) & time(t_2) & t_1 < t_2 \\ happens(e_1, t_1). & act(e_1, load). & \\ happens(e_2, t_2). & act(e_2, shoot). & \end{array}$$

would indicate that the given actions and events are the only ones. Finally, an observation about the world might be that at time t_2, the turkey is not walking:

$$\neg holds(walking, t_2)$$

This summarises the different types of formulae typically encountered in an Event Calculus specification.

The meaning of an Event Calculus theory is given by its justification semantics. The predicates *holds*, *clipped*, *initiates* and *terminates* are defined predicates. As seen in the example, the "primitive" predicates *initially*, *happens* and *act* are open, or can alternatively be defined by enumeration if they are completely known, thus asserting a particular scenario. The other primitive predicates, *time* and $<$, are always open. Observe that if *happens* is defined, its definition should contain the clause

$$happens(start, t_0).$$

which was given as a FOL axiom before. Likewise, the definition of *initiates* should contain the clause

$$initiates(start, P) \leftarrow initially(P).$$

The corresponding FOL axioms are then of course redundant.

This completes the formalisation of the Event Calculus. Comparing this formalisation with other ones, in particular with the several variants described in [93], it is worth observing that our variant is closest to the *New Event Calculus* described in that paper. This is due to our formulation of Event Calculus as an *open* logic program with FOL axioms rather than as a classical logic program. As a result, our variant has all the advantages of New Event Calculus described in [93].

4.2.2 Reasoning on Event Calculus Specifications

Several forms of reasoning on temporal domains are usually deemed of importance, in particular temporal projection, diagnosis/postdiction and planning. We illustrate how these forms of reasoning are supported by an abductive procedure like SLDNFA.

The most straightforward form of reasoning is temporal projection: given the specification of an initial state and a sequence of actions, the task is to calculate the evolution of the world, i.e. which fluents hold at which times. This is essentially a deductive task. An example is the classical Yale Shooting Problem: given is the turkey domain, without the *walking* fluent and the *go* action. Effect rules are

$$\begin{array}{lll} initiates(E, loaded) & \leftarrow & act(E, load). \\ terminates(E, loaded) & \leftarrow & act(E, shoot). \\ terminates(E, alive) & \leftarrow & act(E, shoot), happens(E, T), \\ & & holds(loaded, T). \end{array}$$

The following scenario is asserted by program clauses (defining *happens*, *act* and *initially*):

$$\begin{array}{ll} happens(start, t_0). & initially(alive). \\ happens(e_1, t_1). & act(e_1, load). \\ happens(e_2, t_2). & act(e_2, load). \\ happens(e_3, t_3). & act(e_3, shoot). \end{array}$$

and in addition the following axioms on the open predicates *time* and $<$ are given:

$$\begin{array}{ll} time(t_1) & \\ time(t_2) & t_1 < t_2 \\ time(t_3) & t_2 < t_3 \end{array}$$

Any question about the truth value of fluents at times after t_0 can now be answered. For example, the classical question which has caused so many

problems up to the eighties, is if the turkey can be alive after the sequence of events, which can be answered by trying to solve the goal

$$\leftarrow time(t_4), t_3 < t_4, holds(alive, t_4).$$

which in our formalisation fails, as it should.

In the context of incomplete knowledge about some of the primitive predicates, other forms of reasoning are also of interest. One is postdiction, which is relevant when the initial state is not known: the goal in postdiction is to find an initial state which explains certain later observations. This is an abductive task very similar to diagnosis. An example is the Stanford Murder Mystery, in which we are uncertain of the initial state of the gun, but know that the turkey is initially alive and dead after a shot. Definitions for *happens* and *act* are

$$\begin{array}{l} happens(start, t_0). \\ happens(e_1, t_1). \\ act(e_1, shoot). \end{array}$$

while *time*, $<$ and *initially* are open predicates. A FOL axiom on *initially* is

$$initially(alive)$$

The observation to be explained is presented as a goal to SLDNFA. In this case this goal is $\leftarrow t_1 < t_2, \neg holds(alive, t_2)$, and the answer Δ contains for the *initially* predicate:

$$\begin{array}{l} initially(alive). \\ initially(loaded). \end{array}$$

Note that there are several other, equivalent ways of representing this problem: in particular, all observations could be represented as FOL axioms, which is in fact the "correct" representation of the problem as we defined it

$$\begin{array}{c} initially(alive) \\ t_1 < t_2 \\ \neg holds(alive, t_2) \end{array}$$

The goal to be solved is then just $\leftarrow true$, and the same answers are obtained. In general it makes no difference for the solution of a particular problem if FOL axioms are moved from the theory to the goal or vice versa, as $T \cup \{F \wedge G\}$ is consistent if and only if $(T \cup \{F\}) \cup \{G\}$ is consistent. However, from a knowledge representation point of view the represented theory is different, and the "correct" representation depends on the precise

wording of the problem. In particular, the specification and goal we first provided in this example correspond to the wording "A turkey is initially alive. There is a shot with a gun that may or may not be loaded. Can the turkey be dead after the shot, and if so how come ?". In the rest of this thesis, when the precise wording of a problem is not so important (as often with example toy problems) we will allow FOL axioms to be moved to the goal as conjuncts and vice versa.

A third typical type of reasoning on temporal domains, and probably the most interesting, is planning. In a planning problem typically there is a given initial state of the world and a desired end state, and the goal is to derive a sequence of actions which, when executed starting in the initial state, leads to the end state. Another way of formulating this is saying that in a given domain the initial state is specified but the actions that occur are unknown (*happens* and *act* are open predicates). Then one asks if the desired end state is consistent with the theory and under what conditions (i.e. the end state is the goal to be solved). This yields an abduced sequence of actions which entails the desired end state. As an example, assume we are again in the Yale Shooting domain and we want a plan to kill the turkey. The initial state is defined by

$$initially(alive).$$

and the goal to be solved is $\leftarrow t_0 < t, \neg holds(alive, t)$. The most straightforward solution contains in Δ (we omit redundant "$<$" and "*time*" facts):

$$\begin{array}{lll} happens(e_1, t_1). & act(e_1, load). & t_1 < t_2. \\ happens(e_2, t_2). & act(e_2, shoot). & t_2 < t. \end{array}$$

These are the most common forms of reasoning on temporal domains, though we will also study several other forms in this thesis. In particular, in the following chapter we will study counterfactual reasoning in the context of Situation Calculus and Event Calculus. Also in Chapter 6 we will discuss various reasoning issues in different settings.

4.2.3 Support for Event Calculus in SLDNFA

In the SLDNFA procedure specialised support for reasoning on Event Calculus specifications has been implemented. First of all, there is a constraint module for efficiently reasoning with partial orders, in particular on time points. The module delays decisions on the order of two time points until this order becomes relevant. When new information is added, the partial

order is efficiently updated. It is ensured that at any time, all linearisations of the current partial order are consistent with the given information (hence the aforementioned correctness problems with partial orders are avoided).

A second feature is that SLDNFA allows for a specialised search strategy on Event Calculus theories: in theories where not all events are known one can impose an upper bound on the number of events desired in a solution, thus avoiding an infinite search space. Alternatively or in combination with this, an iterative deepening search on the number of events is possible, so that solutions with less events are generated first.

The current implementation of SLDNFA is still a prototype and overall not efficient. This is not fundamentally changed by the above extensions. However, these specialised modules at least defeat the worst sources of inefficiency of reasoning on Event Calculus theories.

Chapter 5

Comparing and Integrating Event and Situation Calculus

5.1 Introduction and Motivation

As we indicated in the previous chapter, the Situation Calculus ([72]) is the most widely used formalism for representing dynamic domains. Its ontology is different from that of the Event Calculus in the following ways.

The notion of action occurrences, or *events*, at certain points in time is central to Event Calculus, and these events determine time intervals during which certain fluents hold. In Situation Calculus the central notions are *actions* and *situations*. A situation corresponds to a snapshot of the world at an instant of time. This has been interpreted in a number of different ways, e.g. in [83] situations are considered to be (hypothetical) periods of time between two actions. In [57] situations are assumed to correspond to time points rather than time periods. In [6] a situation is seen as the set of fluents that holds at a certain instant in time. We will adopt the view of situations as time periods. The set of fluents that holds at an instant in time will be called the *state* of the world at that instant. Actions are the cause of situation transitions: a *result* function is used to map each (action,situation) pair to a new situation resulting from the execution of that action in the old situation.

Although the original versions of Event and Situation Calculus do not look very much alike, later versions tend to show more and more similarities.

As a result, a comparison of the two formalisms has been a topic of interest in recent years.

In [83] Situation Calculus has been compared with the original Event Calculus. Several problems of the original Event Calculus — caused in particular by the notion of time intervals and by the use of predicate completion on all predicates — were pointed out, and Situation Calculus was extended with an Event Calculus-like time line running through a set of *actual* situations, adding a previously absent notion of real time to Situation Calculus.

In [57] Situation Calculus is compared with a more recent, simplified version of Event Calculus which does not suffer from the aforementioned problems of the original Calculus. The similarities of the Situation Calculus and the Event Calculus are highlighted by showing that their frame axioms are equivalent under a number of conditions.

Here we make a more detailed analysis of this relation, addressing a number of important issues that were left unanswered in [57]. In particular, we study the assumptions made in that paper to obtain the equivalence result. We argue that though Situation Calculus and Event Calculus are indeed very similar, this is not the end of the story. Some of the differences are only of a syntactic nature, but other differences have important implications for knowledge representation and reasoning, and therefore require careful consideration.

For example, in [57] additional restrictions are imposed on Situation Calculus. Certain forms of reasoning — in particular counterfactual reasoning about action occurrences — are impossible in the restricted version, where they are possible in the original Situation Calculus. As a result, where a translation of descriptions in the high level action description language $\mathcal{A}$ ([37]) into Situation Calculus has been proven sound and complete in [27], such a translation into the restricted form of Situation Calculus of [57] is no longer possible. This is related to the observation that a translation of $\mathcal{A}$ descriptions into Event Calculus is also impossible.

We therefore propose a general formalism which extends both Situation Calculus and Event Calculus. We establish a clear relation between time points and situations — different from the one in [57] and extending that in [83] — and prove that assuming this relation, both original calculi can be seen as instances of the more general calculus. Hence, the new calculus has all the expressive power of Situation Calculus as well as of Event Calculus. We use this new calculus as a tool for analysing the possibilities and restrictions of the original calculi.

Studying the problem of counterfactual reasoning in more detail, we

observe that not only Event Calculus, but Situation Calculus as well falls short in some cases, in particular when actions with nondeterministic effects are present. This is due to restrictions inherent in the data structure used in Situation Calculus. We show how the new calculus can handle these cases of counterfactual reasoning that neither of the original calculi can handle.

For ease of comparison we consider open logic programming formalisations of both calculi. For Event Calculus, we utilise the version presented in the previous chapter, under OLP completion semantics rather than justification semantics for simplicity reasons. Situation Calculus is more often expressed in classical logic, but can also be written as an open logic program under a suitable completion semantics. Methods similar to predicate completion have also been used in classical logic formulations of Situation Calculus, for example in [88]. The proofs in this chapter are all performed in classical logic, based on the completion of the presented open logic programs.

In section 5.2, we describe Situation Calculus and briefly recall the central axioms of Event Calculus. Section 5.3 motivates a detailed comparison of the calculi by pointing out a crucial difference between them and showing where previous equivalence results fall short. In section 5.4 we present the new general calculus and illustrate its use with an application. Section 5.5 formally relates the new calculus to the original ones. In section 5.6, we point out a problem with nondeterministic actions in Situation Calculus and show how the new calculus handles them. In section 5.7 we conclude with a number of additional issues.

5.2 Formalisation of the Calculi

5.2.1 The Situation Calculus

We present an open logic programming formalisation of the Situation Calculus. The basic concepts are ***situations*** and ***actions***. A situation is defined as a period of time during which there are no actions and no changes in fluent values (the world remains in the same state throughout a situation). Actions are the cause of state transitions: if an action A occurs in a situation S, a new situation $result(A, S)$ begins immediately after the action. $holds(P, S)$ means that fluent P is true in situation S. $initiates(A, S, P)$ ($terminates(A, S, P)$) denotes that if action A occurs in situation S, this initiates (terminates) the fluent P, i.e. immediately after A the fluent is true (false). It is assumed that there is an initial state, called s_0.

The frame axiom of the Situation Calculus can be written as the fol-

lowing definition of *holds*:

$$
\begin{array}{lcl}
holds(P, s_0) & \leftarrow & initially(P). \\
holds(P, result(A, S)) & \leftarrow & initiates(A, S, P). \\
holds(P, result(A, S)) & \leftarrow & holds(P, S), \neg terminates(A, S, P).
\end{array}
$$

The frame axiom reads as follows under completion semantics: a fluent P holds in s_0 if it is initially true, and it holds in a later situation $result(A, S)$ either if the action A leading to that situation initiated P, or if P already held in the previous situation and was not terminated by the most recent action. Otherwise, the fluent does not hold.

In Situation Calculus there are no statements about which actions actually occur: each construct $result(A, S)$ denotes a hypothetical situation which would result if A happened in the (also hypothetical) situation S. Hence, it is possible in Situation Calculus to talk at the same time about for example $result(a, S)$ and $result(b, S)$. Both are hypothetical situations which could result from S, and it does not matter if the actions leading to them "really" occur or not. All situations which can be reached from the initial one after any sequence of actions, exist in the tree of situations. For this reason Situation Calculus is said to incorporate a *branching* time topology.

A Situation Calculus description in general consists of the above clauses plus a number of domain dependent clauses defining *initially*, *initiates* and *terminates*. These definitions are also completed. If it is not completely known, the *initially* predicate may be left open instead. Any number of FOL formulae can be added to the theory.

In for example [89] and [83] a second order induction axiom on situations is added to the Situation Calculus. The axiom represents that the situations that can be reached from the initial situation by executing a finite sequence of actions are the only situations that exist. Under justification semantics, a similar axiom[1] is implied by the following definition of situations:

$$
\begin{array}{lcl}
situation(s_0). & & \\
situation(result(A, S)) & \leftarrow & situation(S),\ action(A).
\end{array}
$$

assuming a domain dependent type predicate $action/1$ for actions.

We do not use justification semantics in this chapter, as completion semantics provides an immediate mapping to FOL, which facilitates theorem proving, and as the additional power of justification semantics is not relevant to this discussion except for the fact that it entails the induction axiom

[1] The exact formalisation differs from the one in [83] due to our use of type predicates instead of a sorted logic, but the semantics is the same.

on situations. Hence we simply complete the above definition for situations and represent the induction axiom explicitly by the following second order logic formula:

$$\forall\Phi : [(\forall S : (situation(S) \rightarrow \Phi(S))) \leftarrow$$
$$(\Phi(s_0) \wedge$$
$$\forall A, S : [(\Phi(S) \wedge action(A) \wedge situation(S)) \rightarrow \Phi(result(A, S))]])]$$

Formulations of the Situation Calculus in classical logic often use one predicate *abnormal* instead of *initiates* and *terminates*, not distinguishing positive changes in truth value from negative ones ([6], [72]). Since this distinction is explicit in Event Calculus, we make it explicit in Situation Calculus as well to facilitate comparison. Moreover, as indicated in [57], such distinction results in a more precise and therefore more "meaningful" theory.

The following axioms and clauses summarise the formalisation of the Situation Calculus:

$$\begin{array}{lll} holds(P, s_0) & \leftarrow & initially(P). \\ holds(P, result(A, S)) & \leftarrow & initiates(A, S, P). \\ holds(P, result(A, S)) & \leftarrow & holds(P, S), \neg terminates(A, S, P). \end{array}$$

$$\begin{array}{lll} situation(s_0). & & \\ situation(result(A, S)) & \leftarrow & situation(S), action(A). \end{array}$$

$$\forall\Phi : [(\forall S : (situation(S) \rightarrow \Phi(S))) \leftarrow$$
$$(\Phi(s_0) \wedge$$
$$\forall A, S : [(\Phi(S) \wedge action(A) \wedge situation(S)) \rightarrow \Phi(result(A, S))]])]$$

$$O = \{(initially)\}$$

5.2.2 The Event Calculus

For a detailed discussion of the Event Calculus we refer to the previous chapter. We recall the general axioms and add some where appropriate for this discussion. The defined predicates are *holds*, *initiates* and *terminates*, the open ones *time*, $<$, *happens*, *act* and *initially*. First of all we have the frame axiom

$$\begin{array}{lll} holds(P, T) & \leftarrow & happens(E_1, T_1), T_1 < T, initiates(E_1, P), \\ & & \neg clipped(T_1, P, T). \\ clipped(T_1, P, T) & \leftarrow & happens(E_2, T_2), T_1 < T_2, T_2 < T, \\ & & terminates(E_2, P). \end{array}$$

and the restrictions on time

$$\begin{gathered}\neg((T_1 < T_2) \wedge (T_2 < T_1)) \\ ((T_1 < T_2) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_3) \\ (time(T_1) \wedge time(T_2)) \rightarrow [(T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)] \\ (T_1 < T_2) \rightarrow (time(T_1) \wedge time(T_2))\end{gathered}$$

Since the Situation Calculus also incorporates an initial situation, we assume in Event Calculus a first event *start* at t_0:

$$\begin{gathered}time(t_0) \\ happens(start, t_0) \\ happens(E, T) \rightarrow ((t_0 < T) \vee (E = start))\end{gathered}$$

This event initiates the fluents that are initially true, as represented by the clause

$$initiates(start, P) \leftarrow initially(P).$$

We adopt the convention that no more than one event can occur at one point in time. Each event can also only happen once:

$$(happens(E, T) \wedge happens(E^*, T^*)) \rightarrow ((E = E^*) \leftrightarrow (T = T^*))$$

Like in Situation Calculus, we introduce a domain dependent predicate $action/1$, and we impose the constraint

$$act(E, A) \rightarrow action(A)$$

We also assume there is at most one action associated with each event:

$$(act(E, A_1) \wedge act(E, A_2)) \rightarrow A_1 = A_2$$

This axiom can be omitted to allow for simultaneous actions, but in Situation Calculus simultaneous actions are usually disallowed, so that we will disregard them in this chapter.

The following axioms and clauses summarise the formalisation of the Event Calculus we will use here:

$$
\begin{array}{lcl}
holds(P,T) & \leftarrow & happens(E_1,T_1), T_1 < T, initiates(E_1,P), \\
 & & \neg clipped(T_1,P,T). \\
clipped(T_1,P,T) & \leftarrow & happens(E_2,T_2), T_1 < T_2, T_2 < T, \\
 & & terminates(E_2,P). \\
 & & \\
initiates(start,P) & \leftarrow & initially(P).
\end{array}
$$

$$\neg((T_1 < T_2) \wedge (T_2 < T_1))$$
$$((T_1 < T_2) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_3)$$
$$(time(T_1) \wedge time(T_2)) \rightarrow [(T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)]$$
$$(T_1 < T_2) \rightarrow (time(T_1) \wedge time(T_2))$$

$$(happens(E,T) \wedge happens(E^*,T^*)) \rightarrow ((E = E^*) \leftrightarrow (T = T^*))$$
$$act(E,A) \rightarrow action(A)$$
$$(act(E,A_1) \wedge act(E,A_2)) \rightarrow A_1 = A_2$$

$$time(t_0)$$
$$happens(start,t_0)$$
$$happens(E,T) \rightarrow ((t_0 < T) \vee (E = start))$$

$$O = \{(initially), (time), (happens), (act), <\}$$

5.2.3 An Example Problem Domain

For our examples we will use the simple and well-known problem domain of the Yale Shooting Problem, which we have already used in the previous chapter. Recall that the important fluents in this domain are *loaded*, indicating that a gun is loaded, and *alive*, indicating that a turkey is alive. Basic actions are *load* (with the effect that the gun becomes loaded), *wait* (which has no effect), and *shoot* (which unloads the gun and kills the turkey if the gun was loaded). The *action* predicate is then defined as

$$action(A) \leftrightarrow [(A = load) \vee (A = wait) \vee (A = shoot)]$$

In Situation Calculus, the effects of these actions are represented by

$$
\begin{array}{lcl}
initiates(A,S,loaded) & \leftarrow & A = load. \\
terminates(A,S,loaded) & \leftarrow & A = shoot. \\
terminates(A,S,alive) & \leftarrow & A = shoot, holds(loaded,S).
\end{array}
$$

The Event Calculus representation of the domain knowledge is

$$\begin{array}{lll} initiates(E, loaded) & \leftarrow & act(E, load). \\ terminates(E, loaded) & \leftarrow & act(E, shoot). \\ terminates(E, alive) & \leftarrow & act(E, shoot),\ happens(E, T), \\ & & holds(loaded, T). \end{array}$$

We will use examples in this problem domain to illustrate and clarify our results.

5.3 Counterfactual Reasoning in Situation Calculus and Event Calculus

As indicated before, in [57] slightly modified versions of Situation Calculus and Event Calculus were shown to be equivalent. The main motivation for our research on this topic was an unexpected problem encountered during an attempt to provide a transformation from $\mathcal{A}$ theories ([37]) to Event Calculus. Where a transformation of $\mathcal{A}$ to Situation Calculus was established and proven sound and complete in [27], our proposed transformation to Event Calculus was incorrect. A detailed analysis pointed out that the problem could not be fixed by modifying the transformation, but was inherent to the Event Calculus, and in particular to its linear time structure. Evidently, the question arose how this related to the equivalence result in [57].

We first illustrate the problem. It concerns the representation of counterfactual statements of the form "If A had happened, then B would have held". Such statements can be correctly represented in formalisms with a *branching* time structure, where one can simultaneously talk about several possible evolutions of the world. Examples of such formalisms are $\mathcal{A}$ and Situation Calculus.

At first sight, counterfactuals can be handled equally well using a form of abductive reasoning on incomplete knowledge in a linear time theory, but this intuition is incorrect. We clarify the point with an example.

Assume we initially have a living turkey, and there is no information on the initial state of the gun. In this situation, a shoot event occurs. We also know that if instead of shooting we had simply waited, the gun would have been loaded afterward. The question is then: can the turkey be alive after the *shoot* event ?

The intended answer is no: since we know that the gun would have been loaded if we had waited instead of shot, we can conclude that it must already have been loaded before the wait event (in the initial situation),

since waiting cannot have loaded the gun. Therefore, a shot in the initial situation should have killed the turkey.

An OLP representation of this problem in Situation Calculus looks as follows. The initial situation is described by the predicate *initially*. This predicate is open as we have no complete information on it. However, there is partial information represented by the FOL axiom

$$initially(alive)$$

We represent the knowledge about the counterfactual situation by a FOL axiom

$$holds(loaded, result(wait, s_0))$$

This axiom constrains the possible values of the open *initially* predicate. The open logic program *without* the FOL axiom has two models: one in which *initially(loaded)* is true and one in which it is false. In the latter model, we know that $\neg holds(loaded, s_0)$, and therefore also that $\neg holds(loaded, result(wait, s_0))$. This is inconsistent with the FOL axiom, so that model of the program clauses is not a model of the entire theory. The only valid model is therefore the one in which *initially(loaded)* is true. In that model, we find $holds(loaded, s_0)$, from which we can obtain the intended result

$$\neg holds(alive, result(shoot, s_0)).$$

using the clauses for termination of fluents.

In Event Calculus, the representation which springs to mind is the following: since we want to reason about different sequences of events, we assume incomplete knowledge on events and their order. We declare *happens* and *act* as well as $<$ open. Representing knowledge about a counterfactual situation is done in a way similar to that in Situation Calculus, by using an axiom of the form "if this sequence of events happens, then this formula will hold afterward". To simplify notation, we first define a new predicate *int_events/2*. $int_events(T, T^*)$ indicates that one or more intermediate events occur between T and T^*.

$$int_events(T, T^*) \leftarrow happens(E', T'), (T < T'), (T' < T^*).$$

Like in Situation Calculus, we have an open *initially* predicate, and our knowledge on the initial situation is represented by the FOL axiom

$$initially(alive)$$

Our knowledge about the counterfactual evolution of the world is represented by

$$(happens(e_1, t_1) \land act(e_1, wait) \land \neg int_events(t_0, t_1)) \\ \rightarrow \forall T : (t_1 < T \land \neg int_events(t_1, T)) \rightarrow holds(loaded, T)$$

and we can express our query as

$$(happens(e_2, t_2) \land act(e_2, shoot) \land \neg int_events(t_0, t_2)) \\ \rightarrow \forall T : (t_2 < T \land \neg int_events(t_2, T)) \rightarrow holds(alive, T))$$

But this is not a correct representation: due to the linear time constraint in Event Calculus, the lefthandside of the axiom and the lefthandside of the query can never evaluate to true in the same interpretation. Either e_2 or e_1 can follow *start* without intermediate events, but not both. Therefore, the axiom has no influence on any useful answers to our query (answers in which the query's lefthandside is true), where it should have provided us with additional information. In fact, we have not been able to model the counterfactual statement *if we had waited, the gun would have been loaded*, but have only approximated it modeling *if we have waited, the gun was loaded afterward.*

The essence of the problem is the following: for each sequence of events, our theory has at least one model in which that sequence occurs, as required. However, within *one* model, only *one* sequence of events exists. Hence, combining information about two unrelated sequences of events is not possible within the formalism, because one sequence always excludes the other.

It should be mentioned that using meta-reasoning on the formalism it is possible to reach the desired conclusions. This is due to the fact that meta-reasoning allows one to consider different object-level models in one meta-level model. As a result, if we restrict time to a linear order we can still perform counterfactual reasoning by meta-reasoning on the formalism.

On the other hand, in Situation Calculus we find all different sequences (branches) of events in one model. This allows us to combine information about these branches, and hence to perform counterfactual reasoning without resorting to a meta-level.

The previous example shows that Situation Calculus can be used to model problems that Event Calculus cannot handle. This is apparently in contradiction with the equivalence result in [57].

The reason for the paradoxical results lies in the modifications applied to Situation Calculus in [57] to prove the equivalence with Event Calculus. To the frame axiom of Situation Calculus, "happens(A,S)" atoms are added,

resulting in the axiom

$$\begin{aligned} holds(P, result(A, S)) &\leftarrow happens(A, S), initiates(A, S, P). \\ holds(P, result(A, S)) &\leftarrow happens(A, S), holds(P, S), \\ &\quad\ \neg terminates(A, S, P). \end{aligned}$$

This addition seems of little consequence, but it has the important effect that the frame axiom is now only applicable to situations resulting from explicitly asserted sequences of actions, like in Event Calculus. Moreover, the FOL axiom

$$(happens(A_1, S) \wedge happens(A_2, S)) \rightarrow A_1 = A_2$$

is added, indicating that only one sequence of actions can exist. This is the counterpart of the linear time constraint in Event Calculus, and it leads to the problems for counterfactual reasoning described above.

In short, the modified Situation Calculus is indeed equivalent to the Event Calculus under the appropriate assumptions, but it is strictly less expressive than the original Situation Calculus. Our goal in this chapter is to slightly extend the Event Calculus and to establish a more general relation between the non-restricted calculi.

The approach we take differs in one other important point from the one in [57]: in the relation between time points and situations. In [57] situations are assumed to correspond to one time point, whereas we consider a situation to be a set of time points, more specifically a set of time points between two actions, like in [83]. This point of view fits in more naturally with the concept of instantaneous actions, which we have adopted earlier.

5.4 A Generalised Calculus

The new calculus we present below extends both Situation Calculus and Event Calculus. Basically, we start with Event Calculus and extend it with branching time. After that, we will define situations and relate the new calculus to Situation Calculus.

5.4.1 The New Calculus

In our new calculus, $event(E, T)$ indicates the occurrence of event E at the hypothetical time point T. The distinction between $event/2$ and the $happens/2$ predicate of Event Calculus is due to the branching time aspect we introduce here. $holds(P, T)$ means P holds at T. *initially*, *initiates*, *terminates*, *act*, *time* and $<$ have the same meaning as in Event Calculus.

The frame axiom is identical to that of Event Calculus with *happens/2* renamed to *event/2* :

$$\begin{array}{lcl} holds(P,T) & \leftarrow & event(E_1,T_1), T_1 < T, initiates(E_1,P), \\ & & \neg clipped(T_1,P,T). \\ clipped(T_1,P,T) & \leftarrow & event(E_2,T_2), T_1 < T_2, T_2 < T, \\ & & terminates(E_2,P). \end{array}$$

The difference with Event Calculus lies in the time structure: instead of the linear time constraints of Event Calculus, we introduce weaker constraints ensuring a branching time structure.

$$\neg((T_1 < T_2) \wedge (T_2 < T_1))$$
$$((T_1 < T_2) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_3)$$
$$((T_1 < T_3) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)$$

$$(T_1 < T_2) \rightarrow (time(T_1) \ \wedge \ time(T_2))$$

Like in Event Calculus, events correspond to exactly one time point and time points to at most one event. We assume a type predicate for actions *action/1*, which is domain dependent. We impose that at least one action exists. Like in Event and Situation Calculus, we also exclude simultaneous actions.

$$(event(E,T) \ \wedge \ event(E^*,T^*)) \rightarrow ((E = E^*) \leftrightarrow (T = T^*))$$
$$\exists A : action(A)$$
$$act(E,A) \rightarrow action(A)$$
$$(act(E,A_1) \ \wedge \ act(E,A_2)) \rightarrow A_1 = A_2$$

We introduce an initial event at t_0:

$$\begin{array}{l} time(t_0) \\ event(start, t_0) \\ event(E,T) \rightarrow ((t_0 < T) \vee (E = start)) \end{array}$$

and relate *initially* to *initiates* by including the following clause in the definition of *initiates*:

$$initiates(start, P) \ \leftarrow \ initially(P).$$

Finally, we need a notion of presence or absence of intermediate events between two events. We define the predicate *int_events* as

$$int_events(T,T^*) \leftarrow event(E',T'), (T < T'), (T' < T^*).$$

The following clauses and axioms summarise the given formalisation:

$$\begin{array}{lll} holds(P,T) & \leftarrow & event(E_1,T_1), T_1 < T, initiates(E_1,P), \\ & & \neg clipped(T_1,P,T). \\ clipped(T_1,P,T) & \leftarrow & event(E_2,T_2), T_1 < T_2, T_2 < T, \\ & & terminates(E_2,P). \\ & & \\ initiates(start,P) & \leftarrow & initially(P). \\ & & \\ int_events(T,T^*) & \leftarrow & event(E',T'), (T < T'), (T' < T^*). \end{array}$$

$$\neg((T_1 < T_2) \wedge (T_2 < T_1))$$
$$((T_1 < T_2) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_3)$$
$$((T_1 < T_3) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)$$
$$(T_1 < T_2) \rightarrow (time(T_1) \ \wedge \ time(T_2))$$

$$(event(E,T) \ \wedge \ event(E^*,T^*)) \rightarrow ((E = E^*) \leftrightarrow (T = T^*))$$
$$\exists A : action(A)$$
$$act(E,A) \rightarrow action(A)$$
$$(act(E,A_1) \ \wedge \ act(E,A_2)) \rightarrow A_1 = A_2$$

$$time(t_0)$$
$$event(start,t_0)$$
$$event(E,T) \rightarrow ((t_0 < T) \vee (E = start))$$

$$O = \{(initially), (time), (event), (action), (act), <\}$$

The meaning of our theory is given by the completion of the above open logic program, plus possibly definitions for *initially*, *initiates* (partially given in terms of *initially* above), *terminates*, *event*, *time* and *act* (and any predicates occurring in these clauses). Some of the predicates (any except *initiates* and *terminates*) may be declared open instead of defined. Any FOL axioms can of course be added to the theory.

5.4.2 Application

We illustrate how we can use the new formalism to represent the counterfactual reasoning problem we encountered earlier, and which we failed to model in Event Calculus.

To summarise the example again: initially there is a living turkey and a gun which may or may not be loaded. Then, a shoot event occurs. We know that, if we had waited instead of shot, the gun would have been loaded afterward. The question is if the turkey can be alive after the shot

(it should not).

As before, *initially* is open. One FOL axiom describes our knowledge on the initial state of the world:

$$initially(alive)$$

We assume incomplete knowledge on *time* and $<$, and express knowledge about these predicates by FOL axioms. We do, however, for the sake of simplicity, assume complete knowledge on (real or counterfactual) relevant events and actions, so we can define these by enumeration. The following facts assert the existence of the events we want to take into account.

$$\begin{array}{ll} event(e_1, t_1). & act(e_1, shoot). \\ event(e_2, t_2). & act(e_2, wait). \end{array}$$

These events occur in mutually exclusive evolutions of the world, so they are on two different branches in the time structure. The axioms

$$\begin{array}{cc} time(t_1) & time(t_2) \\ \neg(t_1 < t_2) & \neg(t_2 < t_1) \end{array}$$

represent this knowledge.

Observations about hypothetical evolutions of the world are in general represented by FOL axioms. In this case we get the axiom

$$\forall T : ((t_2 < T) \wedge \neg int_events(t_2, T) \rightarrow holds(loaded, T))$$

This axiom is only satisfied in models in which $initially(loaded)$ holds. In those models, $holds(loaded, t_1)$ is true, which implies $terminates(e_1, alive)$. Therefore,

$$\forall T : ((t_1 < T) \wedge \neg int_events(t_1, T) \rightarrow \neg holds(alive, T))$$

holds in all models of our theory, which is the intended result.

This application shows that counterfactual reasoning is possible in the new formalism, where it is not in Event Calculus. The addition of branching time is responsible for the gain in expressive power.

5.4.3 Introducing Situations

The open logic program given in section 5.4.1 forms the essence of our new calculus. Now we extend this theory with a number of additional concepts that correspond to the concepts in Situation Calculus.

First of all, we introduce situations. We define a situation to be the set of all time points that are later than a certain event (the starting event of

the situation), and such that there are no other events between the starting event and the time point itself. We call the initial situation s_0, and use the term $result(E, A, S)$ to denote the situation which is started by the event E, with associated action A, occurring in situation S. Note that A and S are uniquely determined by E, but not vice versa. For our discussion it is appropriate to include all three parameters in the situation name.

We get the following inductive definition:

$$\begin{array}{ll} s_0 = & \{T \mid (t_0 < T) \wedge \neg int_events(t_0, T)\} \\ result(E, A, S) = & \{T \mid \exists T' : T' \in S \wedge event(E, T') \wedge act(E, A) \\ & \wedge (T' < T) \wedge \neg int_events(T', T)\} \end{array}$$

which we express in the following clauses for the *member* predicate:

$$\begin{array}{lll} member(T, s_0) & \leftarrow & t_0 < T, \neg int_events(t_0, T)) \\ member(T, result(E, A, S)) & \leftarrow & event(E, T'), member(T', S), \\ & & act(E, A), T' < T, \neg int_events(T', T). \end{array}$$

where $member(T, S)$ denotes $T \in S$.

We can then inductively define a type predicate for situations : s_0 is a situation, and $S = result(E, A, S)$ is a situation if S is a situation, E an event occurring at a time point belonging to S and A the action associated with that event. This is represented by the completion of

$$\begin{array}{lll} situation(s_0). & & \\ situation(result(E, A, S)) & \leftarrow & situation(S), action(A), \\ & & member(T, S), event(E, T), act(E, A). \end{array}$$

with in addition the induction axiom

$$\begin{array}{c} \forall \Phi : [(\forall S : (situation(S) \rightarrow \Phi(S))) \leftarrow \\ (\Phi(s_0) \wedge \forall A, S, E, T : \\ [(\Phi(S) \wedge situation(S) \wedge member(T, S) \wedge event(E, T) \wedge act(E, A)) \\ \rightarrow \Phi(result(E, A, S))])] \end{array}$$

Having introduced situations, we define what it means for a fluent to hold in a situation: we say that a fluent holds in a situation if and only if it holds at all time points belonging to that situation. We use the predicate *holds_in*/2 to express the truth value of fluents in a situation.

$$holds_in(P, S) \leftrightarrow \forall T : (member(T, S) \rightarrow holds(P, T))$$

In general, it is sufficient to check only the truth value at one time point: we can prove that a fluent holds at all time points in a situation if it holds at at least one. This is expressed in the following theorem:

Theorem 5.4.1

$$\begin{array}{c}\forall S : situation(S) \rightarrow \\ ([\exists T : member(T, S) \ \wedge \ holds(P, T)] \rightarrow \\ [\forall T : member(T, S) \rightarrow holds(P, T)])\end{array}$$

Proof:
Assume S is a situation. We know that

$$\exists T : member(T, S) \ \wedge \ holds(P, T)$$

and need to prove $\forall T : member(T, S) \rightarrow holds(P, T)$. To this end, we first rewrite the above formula using the definition of *holds* :

$$\begin{array}{l}\exists T, E', T' : member(T, S) \ \wedge \ event(E', T') \ \wedge \ T' < T \ \wedge \ initiates(E', P) \\ \qquad \wedge \neg clipped(T', P, T)\end{array}$$

We distinguish two cases: a situation S either has the form $result(E, A, S')$ as described in the definition, or is s_0. First assume $S = s_0$: the above formula then reads

$$\begin{array}{c}[\exists T, E', T' : (S = s_0 \ \wedge \ member(T, s_0) \ \wedge \ event(E', T') \ \wedge \ T' < T \ \wedge \\ initiates(E', P) \ \wedge \ \neg clipped(T', P, T))]\end{array}$$

We derive from $member(T, s_0)$ that $t_0 < T$ and $\neg int_events(t_0, T)$. Since $T' < T$ and $t_0 < T$ hold, one of $T' < t_0$, $t_0 < T'$ or $t_0 = T'$ must be true. The first formula is inconsistent with the axiom that there are no events before t_0, the second formula together with $T' < T$ is in contradiction with $\neg int_events(t_0, T)$. Therefore $t_0 = T'$, and we can derive from the above disjunction that

$$\begin{array}{c}[\exists T, E', T' : (S = s_0 \ \wedge \ T' = t_0 \ \wedge \ E' = start \ \wedge \ t_0 < T \ \wedge \\ \neg int_events(t_0, T) \ \wedge \ event(start, t_0) \ \wedge \ initiates(start, P) \\ \wedge \neg clipped(t_0, P, T)]\end{array}$$

which implies

$$S = s_0 \ \wedge \ initiates(start, P)$$

From this, it follows that P holds for each member of s_0. Indeed, assume that $member(T^+, s_0)$: we find for each such T^+

$$t_0 < T^+ \ \wedge \ \neg int_events(t_0, T^+) \ \wedge \ event(start, t_0) \ \wedge \ initiates(start, P)$$

so, because $clipped(T, P, T')$ implies $int_events(T, T')$, it follows that

$$t_0 < T^+ \ \wedge \ \neg clipped(t_0, P, T^+) \ \wedge \ event(start, t_0) \ \wedge \ initiates(start, P)$$

and therefore $holds(P, T^+)$.

Now take the second case: assume $S = result(E^*, A, S^*)$ is a situation. From

$$\exists T, E', T' : member(T, S) \land event(E', T') \land T' < T \land \\ initiates(E', P) \land \neg clipped(T', P, T)$$

we derive, using the definition of *situation* and the Free Equality theory :

$$\exists T, E', T', E^*, T^*, A, S^* : \\ (S = result(E^*, A, S^*) \land member(T, result(E^*, A, S^*)) \\ \land situation(S^*) \land member(T^*, S^*) \land event(E^*, T^*) \land act(E^*, A) \\ \land event(E', T') \land T' < T \land initiates(E', P) \land \neg clipped(T', P, T))$$

which we rewrite using the definition of *member* to

$$\exists T, E', T', E^*, T^*, A, S^* : \\ (S = result(E^*, A, S^*) \land T^* < T \land \neg int_events(T^*, T) \land \\ situation(S^*) \land member(T^*, S^*) \land event(E^*, T^*) \land act(E^*, A) \land \\ event(E', T') \land T' < T \land initiates(E', P) \land \neg clipped(T', P, T))$$

We now use the knowledge that $T' < T$ and $T^* < T$. Given our FOL axioms on the time relation this implies that either $T' < T^*$ or $T' = T^*$ must hold ($T^* < T'$ cannot hold because of $\neg int_events(T^*, T)$):

$$[\exists T, E', T', E^*, T^*, A, S^* : \\ (S = result(E^*, A, S^*) \land T^* < T \land member(T^*, S^*) \\ \land \neg int_events(T^*, T) \land event(E^*, T^*) \land act(E^*, A) \land \\ event(E', T') \land T' < T^* \land initiates(E', P) \land \neg clipped(T', P, T))] \\ \lor \\ [\exists T, E^*, T^*, A, S^* : \\ (S = result(E^*, A, S^*) \land T^* < T \land \neg int_events(T^*, T) \land \\ member(T^*, S^*) \land event(E^*, T^*) \land act(E^*, A) \land initiates(E^*, P) \land \\ \neg clipped(T^*, P, T))]$$

Now, what we must prove is that for each member of S, P holds. So assume

$member(T^+, S)$. We add $member(T^+, S)$ to both disjuncts:

$$\begin{gathered}[\exists T, E', T', E^*, T^*, A, S^* : \\ (S = result(E^*, A, S^*) \wedge T^* < T \wedge member(T^*, S^*) \\ \wedge \neg int_events(T^*, T) \wedge event(E^*, T^*) \wedge act(E^*, A) \wedge \\ event(E', T') \wedge T' < T^* \wedge initiates(E', P) \wedge \neg clipped(T', P, T)) \\ \wedge member(T^+, S)] \\ \vee \\ [\exists T, E^*, T^*, A, S^* : \\ (S = result(E^*, A, S^*) \wedge T^* < T \wedge \neg int_events(T^*, T) \wedge \\ member(T^*, S^*) \wedge event(E^*, T^*) \wedge act(E^*, A) \wedge initiates(E^*, P) \wedge \\ \neg clipped(T^*, P, T)) \wedge member(T^+, S)]\end{gathered}$$

and rewrite them using the definition of member

$$\begin{gathered}[\exists T, E', T', E^*, T^*, A, S^*, T'' : \\ (S = result(E^*, A, S^*) \wedge T^* < T \wedge member(T^*, S^*) \\ \wedge \neg int_events(T^*, T) \wedge event(E^*, T^*) \wedge act(E^*, A) \wedge \\ event(E', T') \wedge T' < T^* \wedge initiates(E', P) \wedge \neg clipped(T', P, T)) \\ \wedge event(E^*, T'') \wedge member(T'', S^*) \wedge T'' < T^+ \wedge \\ \neg int_events(T'', T^+)] \\ \vee \\ [\exists T, E^*, T^*, A, S^*, T'' : \\ (S = result(E^*, A, S^*) \wedge T^* < T \wedge \neg int_events(T^*, T) \wedge \\ member(T^*, S^*) \wedge event(E^*, T^*) \wedge act(E^*, A) \wedge initiates(E^*, P) \wedge \\ \neg clipped(T^*, P, T)) \\ \wedge event(E^*, T'') \wedge member(T'', S^*) \wedge T'' < T^+ \wedge \\ \neg int_events(T'', T^+)]\end{gathered}$$

As events can only be associated with one time point, $T'' = T^*$ must hold, so we can simplify the formula (also omitting some conjuncts) to

$$\begin{gathered}[\exists T, E', T', E^*, T^* : \\ (event(E^*, T^*) \wedge member(T^*, S^*) \wedge act(E^*, A) \wedge \\ T^* < T^+ \wedge \neg int_events(T^*, T^+) \wedge T^* < T \wedge \neg int_events(T^*, T) \wedge \\ event(E', T') \wedge T' < T^* \wedge initiates(E', P) \wedge \neg clipped(T', P, T))] \\ \vee \\ [\exists T, E^*, T^* : \\ (event(E^*, T^*) \wedge member(T^*, S^*) \wedge act(E^*, A) \wedge \\ T^* < T^+ \wedge \neg int_events(T^*, T^+) \wedge T^* < T \wedge \neg int_events(T^*, T) \wedge \\ initiates(E^*, P) \wedge \neg clipped(T^*, P, T))]\end{gathered}$$

Now, in the first disjunct we can expand the definition of *clipped*/3: given $T' < T^*$, $T^* < T$ and $event(E^*, T^*)$, $\neg clipped(T', P, T)$ is equivalent to

$$\neg clipped(T', P, T^*) \wedge \neg terminates(E^*, P) \wedge \neg clipped(T^*, P, T)$$

The first two conjuncts of this formula together with $\neg int_events(T^*, P, T^+)$ imply $\neg clipped(T', P, T^+)$. So from the disjunction above it follows that

$$\begin{array}{c}
[\exists E', T', E^*, T^* : \\
(event(E^*, T^*) \wedge member(T^*, S^*) \wedge act(E^*, A) \wedge \\
T^* < T^+ \wedge event(E', T') \wedge T' < T^* \wedge initiates(E', P) \wedge \\
\neg clipped(T', P, T^+))] \\
\vee \\
[\exists T, E', T' : \\
(event(E', T') \wedge member(T', S^*) \wedge act(E', A) \wedge \\
T' < T^+ \wedge \neg int_events(T', T^+) \wedge T' < T \wedge initiates(E', P))]
\end{array}$$

Which leads, in both disjuncts, to the conclusion that

$$holds(P, T^+)$$

This proves the theorem for the case $S = result(E^*, A, S^*)$. Together with the proof for $S = s_0$ above, this completes the proof of the theorem. □

5.4.4 The Application Revisited

The introduction of situations in the new calculus allows us to simplify the representation of the application in section 5.4.2 a little. We recall the clauses and axioms describing the scenario:

$$initially(alive)$$

$$\begin{array}{ll}
event(e_1, t_1). & act(e_1, shoot). \\
event(e_2, t_2). & act(e_2, wait).
\end{array}$$

$$\begin{array}{ll}
time(t_1) & time(t_2) \\
\neg(t_1 < t_2) & \neg(t_2 < t_1)
\end{array}$$

with open *time*, $<$ and *initially* predicates.

The observation

$$\forall T : ((t_2 < T) \wedge \neg int_events(t_2, T) \rightarrow holds(loaded, T))$$

can now be simplified to the equivalent formula

$$holds_in(loaded, result(e_2, wait, s_0))$$

and similarly the formula under consideration

$$\forall T : ((t_1 < T) \wedge \neg int_events(t_1, T) \rightarrow \neg holds(alive, T))$$

can be simplified to

$$\neg holds_in(alive, result(e_1, shoot, s_0))$$

This representation shows greater similarities with the one in Situation Calculus, and is definitely less cumbersome.

5.5 Relation to the Original Calculi

As indicated before, the new calculus is an extension of both Situation Calculus and Event Calculus. We can obtain Situation Calculus or Event Calculus by adding specific constraints to the general calculus, as we prove in this section. These specific constraints represent the essential difference between Situation Calculus and Event Calculus, and we will further study some of their implications further on.

First, we show that Event Calculus is a special case of the new calculus described in section 5.4.1. The only actual difference between the calculi — if we rename the *event* predicate to *happens* or vice versa, which is only a matter of syntax — is in the constraints on $<$.

In Event Calculus, we have an axiom

$$(time(T_1) \wedge time(T_2)) \rightarrow [(T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)] \qquad (1)$$

where the corresponding axiom in the new calculus reads

$$((T_1 < T_3) \wedge (T_2 < T_3)) \rightarrow [(T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)] \qquad (2)$$

As either calculus contains an axiom $(T_1 < T_2) \rightarrow (time(T_1) \wedge time(T_2))$, the lefthandside of (1) is implied by the lefthandside of (2), while their righthandsides are the same. Therefore, (2) is implied by (1): the Event Calculus axiom is strictly stronger than the new axiom. Intuitively, this corresponds to the fact that a time line is one special case of a branching time structure.

So, to obtain Event Calculus, we only need to strengthen one axiom of our calculus, restricting the time tree to a line.

We now show the relation of the new calculus to Situation Calculus. This is less straightforward, partly because the frame axiom is formulated in a different way, but also because a number of assumptions in Situation Calculus are implicit in the data structure. These assumptions have to be made explicit in the new calculus, which will also result in a clearer view on

them. We start from the theory given in section 5.4.3, summarised below

$$
\begin{array}{lcl}
holds(P,T) & \leftarrow & event(E_1,T_1), T_1 < T, initiates(E_1,P), \\
 & & \neg clipped(T_1,P,T). \\
clipped(T_1,P,T) & \leftarrow & event(E_2,T_2), T_1 < T_2, T_2 < T, \\
 & & terminates(E_2,P). \\
initiates(start,P) & \leftarrow & initially(P). \\
 & & \\
int_events(T,T^*) & \leftarrow & event(E',T'), (T < T'), (T' < T^*).
\end{array}
$$

$$
\begin{gathered}
\neg((T_1 < T_2) \wedge (T_2 < T_1)) \\
((T_1 < T_2) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_3) \\
((T_1 < T_3) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2) \\
(T_1 < T_2) \rightarrow (time(T_1) \ \wedge \ time(T_2))
\end{gathered}
$$

$$
\begin{gathered}
(event(E,T) \ \wedge \ event(E^*,T^*)) \rightarrow ((E = E^*) \leftrightarrow (T = T^*)) \\
\exists A : action(A) \\
act(E,A) \rightarrow action(A) \\
(act(E,A_1) \ \wedge \ act(E,A_2)) \rightarrow A_1 = A_2
\end{gathered}
$$

$$
\begin{gathered}
time(t_0) \\
event(start,t_0) \\
event(E,T) \rightarrow ((t_0 < T) \vee (E = start))
\end{gathered}
$$

$$
\begin{array}{lcl}
situation(s_0). & & \\
situation(result(E,A,S)) & \leftarrow & situation(S), \ action(A), \\
 & & member(T,S), \ event(E,T), \\
 & & act(E,A). \\
 & & \\
member(T,s_0) & \leftarrow & t_0 < T \ , \ \neg int_events(t_0,T)) \\
member(T,result(E,A,S)) & \leftarrow & event(E,T'), \ member(T',S), \\
 & & act(E,A), \ T' < T, \\
 & & \neg int_events(T',T).
\end{array}
$$

$$
\begin{gathered}
\forall \Phi : [(\forall S : (situation(S) \rightarrow \Phi(S))) \leftarrow \\
(\Phi(s_0) \ \wedge \ \forall A,S,E,T : [(\Phi(S) \wedge \\
situation(S) \ \wedge \ member(T,S) \ \wedge \ event(E,T) \ \wedge \ act(E,A)) \\
\rightarrow \Phi(result(E,A,S))]])]
\end{gathered}
$$

$$
holds_in(P,S) \leftrightarrow \forall T : (member(T,S) \ \rightarrow \ holds(P,T))
$$

$$
O = \{initially, time, event, action, act, <\}
$$

A very important assumption inherent to Situation Calculus is the following. In Situation Calculus, each term $result(A, S')$ where A is an action and S' is a situation, represents a new situation. All of these terms exist in the theory and therefore all of these situations are implicitly present in Situation Calculus. Moreover, for each action A and situation S', there is exactly one resulting situation $result(A, S')$.

In other words, in Situation Calculus it is assumed that in each (hypothetical) situation each action occurs (in some hypothetical evolution of the world), and leads to exactly one new situation. This is not necessarily true in the new calculus and requires an extra axiom. Since we have defined a situation as the set of time points after a certain event and before any later events, in our new calculus the assumption reads that for each event E and action A, there is exactly one event E^* consisting of the occurrence of A immediately after E:

$$\begin{array}{c}\forall E, T, A : ((event(E,T) \wedge action(A)) \rightarrow \exists E^*, T^* : \\ {[event(E^*, T^*) \wedge (T < T^*) \wedge act(E^*, A) \wedge \neg int_events(T, T^*)]})\end{array} \qquad (ia)$$

$$\begin{array}{c}\forall E, T, E', T', E^*, T^*, A : ((event(E,T) \wedge action(A) \\ \wedge event(E', T') \wedge act(E', A) \wedge T < T' \wedge \neg int_events(T, T') \\ \wedge event(E^*, T^*) \wedge act(E^*, A) \wedge T < T^* \wedge \neg int_events(T, T^*) \\ \rightarrow (E' = E^*))\end{array} \qquad (ib)$$

It is easy to prove that this axiom implies that each action occurs in each situation in some possible evolution of the world, and leads to exactly one new situation:

Lemma 5.5.1
(Lemma 5.5.1.a)

$$\begin{array}{c}\forall S, A : (situation(S) \wedge action(A)) \rightarrow \\ \exists E, T : member(T, S) \wedge event(E, T) \wedge act(E, A)\end{array}$$

(Lemma 5.5.1.b)

$$\begin{array}{c}\forall S, A, E, T, E', T' : ((situation(S) \wedge member(T, S) \wedge event(E, T) \\ \wedge act(E, A) \wedge member(T', S) \wedge event(E', T') \wedge act(E', A)) \\ \rightarrow (E = E')\end{array}$$

Proof:
The lemma follows from axioms (ia) and (ib). We prove part a first: for all S and A, if $situation(S) \wedge action(A)$ then

$$\begin{array}{c}action(A) \wedge \\ ((S = s_0) \vee [\exists E^*, B, S^* : (S = result(E^*, B, S^*) \wedge situation(S^*)) \\ \wedge action(B) \wedge \exists T^* : (member(T^*, S) \wedge event(E^*, T^*) \wedge act(E^*, B))])\end{array}$$

With the fact $event(start, t_0)$ and axiom (ia) this implies

$$[S = s_0 \wedge event(start, t_0) \wedge \exists E, T : (event(E,T) \wedge t_0 < T \wedge act(E, A) \wedge \neg int_events(t_0, T))]$$
$$\vee$$
$$[\exists E, T, E^*, T^*, B, S^* : (S = result(E^*, B, S^*) \wedge action(B) \wedge situation(S^*) \wedge member(T^*, S) \wedge event(E^*, T^*) \wedge act(E^*, B) \wedge event(E, T) \wedge T^* < T \wedge act(E, A) \wedge \neg int_events(T^*, T))]$$

which, using the definition of *member*, leads to

$$\exists E, T : member(T, s_0) \wedge S = s_0 \wedge event(E, T) \wedge act(E, A)$$
$$\vee$$
$$\exists E, T, B, E^*, S^* : [member(T, result(E^*, B, S^*)) \wedge event(E, T) \wedge act(E, A) \wedge S = result(E^*, B, S^*)]$$

and therefore

$$\exists E, T : member(T, S) \wedge event(E, T) \wedge act(E, A)$$

which is what we needed to prove.

The proof of part b is as follows: given S, A, E, T, E', T', assume

$$(situation(S) \wedge member(T, S) \wedge event(E, T) \wedge act(E, A) \wedge member(T', S) \wedge event(E', T') \wedge act(E', A))$$

Again, we use the definition of *situation* to obtain

$$[S = s_0 \vee \exists B, S^*, E^*, T^* : (S = result(E^*, B, S^*) \wedge action(B) \wedge situation(S^*) \wedge member(T^*, S^*) \wedge event(E^*, T^*) \wedge act(E^*, B))] \wedge member(T, S) \wedge event(E, T) \wedge act(E, A) \wedge member(T', S) \wedge event(E', T') \wedge act(E', A))$$

which can be rewritten as

$$[S = s_0 \wedge member(T, S) \wedge event(E, T) \wedge act(E, A) \wedge member(T', S) \wedge event(E', T') \wedge act(E', A)]$$
$$\vee$$
$$[\exists B, S^*, E^*, T^* : (S = result(E^*, B, S^*) \wedge action(B) \wedge situation(S^*) \wedge member(T^*, S^*) \wedge event(E^*, T^*) \wedge act(E^*, B)) \wedge member(T, S) \wedge event(E, T) \wedge act(E, A) \wedge member(T', S) \wedge event(E', T') \wedge act(E', A))]$$

and using the definition of member we get

$$\begin{gathered}
[S = s_0 \ \wedge \ event(start, t_0)] \\
\wedge\, t_0 < T \ \wedge \ \neg int_events(t_0, T) \ \wedge \ event(E, T) \ \wedge \ act(E, A) \wedge \\
t_0 < T' \ \wedge \ \neg int_events(t_0, T') \ \wedge \ event(E', T') \ \wedge \ act(E', A) \\
\vee \\
[\exists B, S^*, E^*, T^* : (S = result(E^*, B, S^*) \ \wedge \ action(B) \ \wedge \ situation(S^*) \\
\wedge\, member(T^*, S^*) \ \wedge \ event(E^*, T^*) \ \wedge \ act(E^*, B)) \\
\wedge\, T^* < T \ \wedge \ \neg int_events(T^*, T) \wedge \ event(E, T) \ \wedge \ act(E, A) \\
\wedge\, T^* < T' \ \wedge \ \neg int_events(T^*, T') \ \wedge \ event(E', T') \ \wedge \ act(E', A))]
\end{gathered}$$

Applying axiom (ib) to either part of the disjunction, we find

$$E = E'$$

which proves the lemma. □

This lemma also implies that to each (action,situation) pair there corresponds exactly one event consisting of that action occurring in that situation. A situation is then completely determined by the previous situation and the last action. Therefore, we can eliminate the event parameter in situation names and reduce the term $result(E, A, S)$ to $result(A, S)$. Our names for situations then coincide with those of Situation Calculus.

Given the above lemma, the induction axiom of section 5.4.3 can be simplified to the Situation Calculus induction axiom

$$\begin{gathered}
\forall \Phi : [(\forall S : (situation(S) \rightarrow \Phi(S))) \leftarrow \\
(\Phi(s_0) \ \wedge \ \forall A, S : [(\Phi(S) \ \wedge \ action(A) \ \wedge \ situation(S)) \\
\rightarrow \Phi(result(A, S))]])]
\end{gathered}$$

which we will use as the basis for our proof.

We are then about ready to prove the equivalence of the frame axioms of Situation Calculus and the new calculus. Now we only need to define the concepts corresponding to the Situation Calculus predicates *initiates*/3 and *terminates*/3 in the new calculus. These predicates indicate when an action executed in a certain situation initiates or terminates a fluent. In terms of events the definitions read as follows (choosing new names *init_s*/3 and *term_s*/3 to avoid confusion with *initiates*/2 and *terminates*/2 for events):

$$\begin{gathered}
\forall A, S, P : (init_s(A, S, P) \leftrightarrow \exists E, T : \\
(member(T, S) \ \wedge \ event(E, T) \ \wedge \ act(E, A) \ \wedge \ initiates(E, P))) \\
\forall A, S, P : (term_s(A, S, P) \leftrightarrow \exists E, T : \\
(member(T, S) \ \wedge \ event(E, T) \ \wedge \ act(E, A) \ \wedge \ terminates(E, P)))
\end{gathered}$$

With all the necessary concepts defined, we can now state our equivalence result. Intuitively, this result states that the predicate *holds_in*, defining the truth value of a fluent in a situation, coincides with the predicate *holds'* defined in terms of *init_s* and *term_s* by the Situation Calculus frame axiom as follows:

$$\begin{array}{lll} holds'(P, s_0) & \leftarrow & initially(P). \\ holds'(P, result(A, S)) & \leftarrow & init_s(A, S, P). \\ holds'(P, result(A, S)) & \leftarrow & holds'(P, S), \neg term_s(A, S, P). \end{array}$$

More formally, our result is the following: assume T is the open logic program given on page 81, with in addition the axioms

$$\forall E, T, A : ((event(E, T) \wedge action(A)) \rightarrow \exists E^*, T^* : [event(E^*, T^*) \wedge (T < T^*) \wedge act(E^*, A) \wedge \neg int_events(T, T^*)])$$

$$\forall E, T, E', T', E^*, T^*, A : ((event(E, T) \wedge action(A) \wedge event(E', T') \wedge act(E', A) \wedge T < T' \wedge \neg int_events(T, T') \wedge event(E^*, T^*) \wedge act(E^*, A) \wedge T < T^* \wedge \neg int_events(T, T^*) \rightarrow (E' = E^*))$$

$$\forall A, S, P : (init_s(A, S, P) \leftrightarrow \exists E, T : (member(T, S) \wedge event(E, T) \wedge act(E, A) \wedge initiates(E, P)))$$
$$\forall A, S, P : (term_s(A, S, P) \leftrightarrow \exists E, T : (member(T, S) \wedge event(E, T) \wedge act(E, A) \wedge terminates(E, P)))$$

and with open *initiates*, *terminates*, *time*, *event*, *action*, *act* and $<$ predicates. Then

Theorem 5.5.1

$$T \models \forall P, S : situation(S) \rightarrow (holds_in(P, S) \leftrightarrow holds'(P, S))$$

To prove this theorem, we first prove a lemma which extends theorem 5.4.1:

Lemma 5.5.2

$$\begin{array}{c} \forall S : situation(S) \rightarrow \\ ([\exists T : member(T, S) \wedge holds(P, T)] \leftrightarrow \\ [\forall T : member(T, S) \rightarrow holds(P, T)]) \end{array}$$

Proof:
The "$\rightarrow$ part" is theorem 5.4.1. The proof of the "$\leftarrow$ part" is straightforward: all that needs to be proven for all S is $\exists T : member(T, S)$, which

follows immediately from lemma 5.5.1.a and the axiom that at least one action exists. □

We now proceed with the proof of theorem 5.5.1.

Proof:

The proof of our theorem is by induction on situations. Using the induction axiom on situations we find that the theorem follows from the formulae

$$\forall P : holds'(P, s_0) \leftrightarrow holds_in(P, s_0)$$

$$\begin{gathered}\forall P, A, S : \\ (situation(S) \wedge action(A)) \rightarrow [(holds'(P, S) \leftrightarrow holds_in(P, S)) \rightarrow \\ (holds'(P, result(A, S)) \leftrightarrow holds_in(P, result(A, S))]\end{gathered}$$

which we prove here.

The first formula (the base case) can be proven as follows: using the definition of *holds_in*, we write $holds_in(P, s_0)$ as

$$\forall T : [(t_0 < T \wedge \neg int_events(t_0, T)) \rightarrow holds(P, T)]$$

and using the definition of *holds* this is equivalent to

$$\begin{gathered}\forall T : [(t_0 < T \wedge \neg int_events(t_0, T)) \rightarrow \exists E', T' : \\ (event(E', T') \wedge T' < T \wedge initiates(E', P) \wedge \neg clipped(T', P, T))]\end{gathered}$$

Because of the precondition $t_0 < T \wedge \neg int_events(t_0, T)$ and the axiom that t_0 is the first time point, we know that $T' = t_0$. The above formula then reads

$$\begin{gathered}\forall T : [(t_0 < T \wedge \neg int_events(t_0, T)) \rightarrow \\ (event(start, t_0) \wedge t_0 < T \wedge initiates(start, P) \wedge \neg clipped(t_0, P, T))]\end{gathered}$$

Provided that $\exists T : (t_0 < T \wedge \neg int_events(t_0, T))$, which follows immediately from axiom (*ia*), this formula implies $initiates(start, P)$. On the other hand, $initiates(start, P)$ implies the above formula, as $\neg clipped(t_0, P, T)$ trivially follows from $\neg int_events(t_0, T)$.

So we find that $holds_in(P, s_0) \leftrightarrow initiates(start, P)$. Of course, we also know that $initiates(start, P)$ is equivalent to $initially(P)$ and therefore to $holds'(P, s_0)$. This completes the proof of the base case.

The proof of the induction step is rather tedious. We state the result here as a new lemma:

Lemma 5.5.3

$$\begin{gathered}T \models \\ \forall P, A, S : \\ [situation(S) \wedge action(A)] \rightarrow [(holds'(P, S) \leftrightarrow holds_in(P, S)) \rightarrow \\ (holds'(P, result(A, S)) \leftrightarrow holds_in(P, result(A, S))]\end{gathered}$$

We refer the reader to Appendix B for the proof.

Given the base case proven above and lemma 5.5.3, our theorem now follows directly from the induction axiom. Hence, the predicate *holds'*/2 defined by the Situation Calculus frame axiom coincides with *holds_in*/2 defined in terms of the *holds* predicate for time points in our new calculus. □

5.6 Restrictions of Situation Calculus in Counterfactual Reasoning

As we argued in section 5.5, the key reason for the success of Situation Calculus —compared to Event Calculus— at counterfactual reasoning was that in each model of a Situation Calculus theory, all relevant real and counterfactual situations were present. On the logical level, the existence of the necessary situations in Situation Calculus is ensured by :

1. the use of the functor *result*/2 to encode situations, combined with the fact that in classical logic a functor represents a total mapping, as stated explicitly by the tautology $\forall A, S_1 : \exists S_2 : result(A, S_1) = S_2$. Our definition of the type predicate *situation*/1 ensures that S_2 is a situation if A is an action and S_1 a situation.

2. the presence of the Free Equality theory, which entails that different situation terms represent different situations.

In this section, we further investigate this central issue for counterfactual reasoning. We do this by introducing a class of temporal domains and showing that representations of these domains in Situation Calculus do *not* allow to derive intended conclusions based on counterfactual information. The reason for this will be shown to be that for these domains, also in Situation Calculus, not all relevant counterfactual situations are present in each model.

Essential for the class of temporal domains under scrutiny is the presence of nondeterministic actions: actions which, when executed in one and the same situation, do not always have the same effects and hence can lead to different resulting states.

There are several different ways in which nondeterministic actions can arise in a temporal domain. The modelled action may be truly nondeterministic, like the radioactive decay of particles, or it may be deterministic but dependent on very small fluctuations in the world or in the way the action is performed, details which are not modelled in the theory. In some

cases, for example when representing the action of rolling a die, these details can not be reasonably modelled. In other cases the domain may be modelled at a high level of abstraction, on which the details that influence the action are not visible. In the latter case the nondeterminism can be eliminated by using a lower, more detailed, level of abstraction.

We start with an example illustrating the representation of nondeterministic actions in Situation Calculus.

Assume a variant of the Yale shooting problem where the *shoot* action has a nondeterministic effect: sometimes the bullet only wounds the turkey and does not kill it (for example because the hunter has waited a split second longer, the gun has been aimed a little differently, the turkey has moved slightly or for some other unknown reason). If we perform *shoot* in a situation in which both *alive* and *loaded* hold, two resulting situations are possible: one in which *alive* holds (and *wounded* as well), and one in which it does not.

The effect of shooting could in this case be modelled by the formula

$$\begin{aligned}&initiates(A, S, wounded) \vee terminates(A, S, alive)\\ &\quad \leftarrow A = shoot \wedge holds(loaded, S) \wedge holds(alive, S)\end{aligned}$$

plus some formulae "completing" the above one (the disjunction in the head, which represents the nondeterminism, cannot be handled by standard predicate completion):

$$\begin{aligned}initiates(A, S, wounded) &\rightarrow A = shoot \wedge holds(loaded, S)\\ &\quad \wedge holds(alive, S)\\ &\quad \wedge \neg terminates(A, S, alive)\\[1em] terminates(A, S, alive) &\rightarrow A = shoot \wedge holds(loaded, S)\\ &\quad \wedge holds(alive, S)\\ &\quad \wedge \neg initiates(A, S, wounded)\end{aligned}$$

A representation which fits better into our formalism and which allows for the use of standard completion, can be obtained from the above one by introducing an open "degree of freedom" predicate. The use of degree of freedom predicates for representing nondeterminism was originally proposed in [28] in the context of Event Calculus. In general, specifications of the above type using disjunctions in the head of clauses can be transformed rather easily into pure OLP specifications with degree of freedom predicates.

We illustrate the method by modeling the above problem domain in OLP.[2] The rules for initiation and termination read:

$$
\begin{array}{lcl}
initiates(A, S, loaded) & \leftarrow & A = load. \\
terminates(A, S, loaded) & \leftarrow & A = shoot. \\
terminates(A, S, alive) & \leftarrow & A = shoot,\ holds(loaded, S), \\
 & & holds(alive, S),\ luck(A, S). \\
initiates(A, S, wounded) & \leftarrow & A = shoot,\ holds(loaded, S), \\
 & & holds(alive, S),\ \neg luck(A, S).
\end{array}
$$

where $luck/2$ is an open predicate indicating whether or not the hunter is lucky (i.e. whether he kills the turkey or not) during a particular instance of the *shoot* action — determined by the situation in which it occurs —.

The actual outcome of the nondeterministic action (i.e. the choice of disjunct in the original effect axiom) depends on this open predicate, on which we have no information. The theory does not entail or exclude either outcome.

Note that the truth value of the open predicate is not dependent on the state of the world at the action's time of occurrence: the predicate should be considered to represent an unknown modifier of the nondeterministic action. It is not a fluent. This is slightly more apparent in the Event Calculus/new calculus representation:

$$
\begin{array}{lcl}
initiates(E, loaded) & \leftarrow & act(E, load). \\
terminates(E, loaded) & \leftarrow & act(E, shoot). \\
terminates(E, alive) & \leftarrow & act(E, shoot),\ event(E, T),\ luck(E), \\
 & & holds(loaded, T),\ holds(alive, T). \\
initiates(E, wounded) & \leftarrow & act(E, shoot),\ event(E, T),\ \neg luck(E), \\
 & & holds(loaded, T),\ holds(alive, T).
\end{array}
$$

where $luck/1$ is an open predicate parameterised with the event it refers to. In the Situation Calculus representation the (action,situation) pair is the equivalent of this event.

The following scenario illustrates some forms of reasoning with nondeterministic actions in Situation Calculus. Given a turkey which is initially alive and a gun which is initially loaded, as represented by

$$
\begin{array}{l}
initially(alive). \\
initially(loaded).
\end{array}
$$

[2] For a more detailed discussion on nondeterministic actions and the use of degree of freedom predicates we refer to Chapter 7.

we want to determine whether the turkey is alive or not after shooting, so if $holds(alive, result(shoot, s_0))$ or its negation are entailed. The only open predicate is $luck/2$, as explained above.

Because $luck/2$ is open, the theory has models in which $luck(shoot, s_0)$ is true as well as models in which it is false. In the former set of models, the formula $holds(alive, result(shoot, s_0))$ is false, in the latter it is true. Therefore, as intended, neither this formula nor its negation are entailed.

Independent observations may give additional information on a scenario. If in the scenario above we had observed that the turkey was dead after we had reloaded the gun, i.e. if $\neg holds(alive, result(load, result(shoot, s_0)))$ had been an additional FOL axiom, the theory would entail $luck(shoot, s_0)$ and therefore $\neg holds(alive, result(shoot, s_0))$.

This briefly illustrates that Situation Calculus is capable of certain forms of reasoning with nondeterministic actions.

However, problems arise when on these nondeterministic actions counterfactual reasoning is performed. The reason for this is that Situation Calculus cannot represent all relevant situations resulting from a nondeterministic action at the same time.

The following scenario illustrates this: we shoot a turkey, which is initially alive, with a gun that may or may not be loaded. We have the additional information that the outcome of shooting under the existing but to us unknown circumstances *could* have been the death of the turkey. Note that no matter whether the gun was loaded or not, the turkey could still be alive after shooting, due to the nondeterminism involved in the action. The information that the turkey *could* be dead after a shot given the existing initial situation indicates that the gun was initially loaded.

We can represent the scenario in Situation Calculus as follows. The *initially* predicate is open, and one FOL axiom about it is given:

$$initially(alive)$$

The *luck* predicate is open as well. The information about the counterfactual outcome of shooting is represented by the axiom

$$\neg holds(alive, result(shoot, s_0))$$

which entails $initially(loaded)$, as intended. However, it also trivially entails the falsehood of

$$holds(alive, result(shoot, s_0))$$

So in Situation Calculus, we reach the unintended conclusion that the turkey would always be dead after shooting.

In a theory correctly representing the above scenario, it is required that more than one situation resulting from a *shoot* action in the initial situation is present in one model. This problem cannot be solved in Situation Calculus, although it can arguably be side-stepped by replacing the *shoot* action type with two different (deterministic) action types, for example *shoot_and_kill* and *shoot_and_wound*. The nondeterminism is then eliminated, which evidently avoids the problem.

As opposed to Situation Calculus, the new calculus allows for a direct representation of nondeterministic actions and for counterfactual reasoning on them. This is due to the fact that the new calculus allows multiple situations to result from the same sequence of actions in the same model. If we can enforce the existence of these different situations, counterfactual reasoning is no longer a problem.

The existence of all possible situations can be guaranteed by FOL axioms. If only deterministic actions are present, axiom *(ia)* of section 5.5 or Lemma 5.5.1.a which is derived from it, are sufficient. In terms of situations, this axiom can also be written as

$$\forall S, A : (situation(S) \wedge action(A)) \rightarrow \exists E : situation(result(E, A, S))$$

When nondeterministic actions are present, this axiom must be extended with specific axioms ensuring that for each nondeterministic action the situation tree contains situations corresponding to all possible outcomes of that action.

As an example, we take the Yale shooting domain with nondeterministic shoot action we modelled earlier. The existence of all possible situations resulting from shooting can be ensured by the following axiom (note that nondeterminism only occurs when both *alive* and *loaded* hold in a situation).

$$\begin{aligned}\forall S : (situation(S) \wedge holds_in(loaded, S) \wedge holds_in(alive, S)) \rightarrow \\ [\exists E : (situation(result(E, shoot, S) \wedge luck(E)) \\ \wedge \exists E' : (situation(result(E', shoot, S) \wedge \neg luck(E'))]\end{aligned}$$

Similar axioms can be added to any domain description for each nondeterministic action it contains.

As an example of counterfactual reasoning on this domain, we represent the above scenario in the new calculus. Undefined predicates are *initially* and *luck* as well as *event*, *act*, *time* and $<$. There is a FOL axiom on *initially*:

$$initially(alive)$$

The information on the counterfactual shoot event can be represented by the axioms

$$\exists E_1 : \quad [situation(result(E_1, shoot, s_0)) \wedge \neg holds_in(alive, result(E_1, shoot, s_0))]$$

We have to check if the turkey can survive a shot under the given circumstances, i.e. if

$$\exists E : situation(result(E, shoot, s_0)) \wedge holds_in(alive, result(E, shoot, s_0))$$

The above axioms $\exists E_1 : \neg holds_in(alive, result(E_1, shoot, s_0))$ and $initially(alive)$ entail $initially(loaded)$. This in turn entails, combined with $initially(alive)$ and the existence of situations axiom for the shoot action, that

$$\exists E : situation(result(E, shoot, s_0)) \wedge \neg luck(E)$$

The initiation and termination rules on the other hand guarantee that for each E

$$situation(result(E, shoot, s_0)) \wedge \neg luck(E) \rightarrow holds_in(alive, result(E, shoot, s_0)) \wedge holds_in(wounded, result(E, shoot, s_0))$$

is true. Combining these formulae we conclude that the turkey can be alive after shooting. Moreover, the theory also entails that if the turkey survives the shot, it will certainly be wounded.

This example shows how the deficiency of Situation Calculus for counterfactual reasoning can be dealt with in a more general calculus, in which the time tree contains more situations.

5.7 Discussion

In this chapter we have presented a new calculus extending Situation Calculus and Event Calculus, and used that calculus as an analysis tool for comparing the original calculi.

Our approach differs in several respects from the comparison made in [57]. First of all, in [57] the similarities of the calculi were highlighted by reducing them to a common core. Here we complement this work, by creating a calculus which extends rather than restricts both original calculi and by highlighting the restrictions needed to obtain either of them from the general calculus. We study the implications of these restrictions, indicating in what aspects the original calculi deviate from the new one and for which types of reasoning they fall short.

Another difference between our comparison and the one in [57] is the way in which time points are related to situations. In [57], there is a one-to-one correspondence between Situation Calculus situations and Event Calculus time points. A situation is seen as a snapshot of the world at one instant in time. However, we favour the view in for example [83], where situations correspond to extended periods of time. Hence our definition of situations as sets of time points.

Our approach also differs from Pinto and Reiter's: in [83] a time line (a path of actual event occurrences) is added to Situation Calculus, running only through one "actual" sequence of situations. In our proposal we treat all situations as equal, creating a tree of (hypothetical) events and time points running through all of the situations instead of only through one sequence of them. The extra expressive power obtained by this embedding of a time line in situations (which, for example, is necessary for modeling continuously changing fluents) is thereby extended to all hypothetical situations instead of to only the actual ones.[3]

As we indicated in the introduction, one of the motivations for this research was our failure to find a sound and complete translation of $\mathcal{A}$ descriptions into Event Calculus. The advantage of such a translation of high-level action languages like $\mathcal{A}$ into temporal reasoning formalisms, is that a more general evaluation of these formalisms is obtained than by only presenting a number of standard examples and handling them in the formalism.

Since the introduction of $\mathcal{A}$, translations of that language into many formalisms have been described, for example in [54] (where the classical logic formalisations of Situation Calculus in [82] and [88], as well as Baker's circumscriptive approach described in [6], are presented as translations from $\mathcal{A}$), [27] (where a translation into open logic programming Situation Calculus is given) and [107] which translates $\mathcal{A}$ into equational logic programming. Soundness and completeness theorems for these translations are given.

As can be expected given the results described in this chapter and in [27], a mapping of $\mathcal{A}$ descriptions into the new calculus is also possible.

Informally, this follows from the following observations: a mapping of $\mathcal{A}$ into open logic programming Situation Calculus is provided in [27]. The Situation Calculus used there differs in one small respect from ours: a

[3] Related to this, it is worth mentioning that our situations do not correspond to time *periods* in a strict sense, due to the fact that we are working with a branching time structure: the sets of time points specified by our definition of situations are not short straight lines, but rather small trees with one event for a root (this root event is not part of the situation) and with each branch either ending in an event or running on forever.

predicate *noninertial*/3 is used instead of *initiates*/3 and *terminates*/3. However, it is not hard to modify the mapping to deal with this difference.

A second slight complication is that we need to integrate the mapping of Situation Calculus into the new calculus with the mapping of $\mathcal{A}$ into Situation Calculus. However, also this problem can be dealt with rather easily. We do not go into the details in this thesis. However, in Chapter 7 we will present our own high-level language $\mathcal{ER}$, which deals with many more issues than $\mathcal{A}$ and its successors, and we will present a mapping of this formalism to OLP Event Calculus.

Regarding the new calculus, some other issues are of interest. For example, in certain cases (natural language processing springs to mind) it may be important to distinguish between actual and hypothetical or counterfactual events. In our formalism there is no distinction between them. However, it is not hard to cure this. We can select one line of time points to be the actual one, for example using the predicate *actual*/1 on time points. This predicate would have to satisfy the following FOL axioms for all T_1 and T_2:

$$actual(T_1) \rightarrow time(T_1)$$
$$(actual(T_1) \wedge actual(T_2)) \rightarrow [(T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)]$$
$$actual(t_0)$$

indicating that all actual time points are time points, that they form a line, and that the initial time point is actual.[4]

In [92] the issue of *narratives* is addressed. Narratives are typically what Event Calculus-like theories represent: a narrative is a course of real events about which there might be incomplete information.[5] In [92] such courses of real events are introduced in Situation Calculus. There is some relation to the work of [83], though the approach is different. The issue of incomplete knowledge on the course of events is addressed explicitly, as well as the issue of simultaneous or overlapping actions (which have a non-zero duration). On the other hand, like in [83] the issue of counterfactual reasoning is not addressed. We feel the paper is complementary to our work.

Throughout the chapter we have stressed the importance of the existence of all relevant situations in each model for counterfactual reasoning.

[4] Obviously, the actual time line corresponds exactly to the time structure of Event Calculus. Event Calculus can be seen as a restriction of the new calculus to only real time points.

[5] In our Event Calculus this would be represented by an open *happens* predicate partially defined by FOL axioms.

In particular, counterfactual reasoning on nondeterministic actions is possible when multiple situations resulting from the same action are present in a situation tree. One related issue we have not explicitly addressed is counterfactual reasoning on the initial state of the world, i.e. the representation of statements like "If the gun had not been loaded initially, the turkey would have been alive after shooting". A correct representation of such statements requires that models contain multiple "possible" initial situations.

This can be achieved in a way very similar to the one we used when dealing with nondeterministic actions: an extra parameter can be included in the *initially* predicate; this parameter should distinguish between several possible initial situations, just like the extra event parameter in situation names distinguishes between different resulting situations from an action. For example, $initially(alive, s_2)$ would represent that *alive* holds in one of the initial situations, s_2. Each initial situation then functions as the root of one tree of situations. Counterfactual reasoning can be performed by combining information on different possible initial situations.

Alternatively, we can adopt a different representation of the initial situation — like in [57] —, assuming the initial situation is the result of a *creation* action which itself occurs in a (pre-initial) situation in which no fluents hold. Then the above solution is a special case of the solution for nondeterministic actions, obtained by making *creation* nondeterministic. The multiple required "initial" situations for counterfactual reasoning are then obtained as the possible outcomes of the *creation* action.

This concludes our comparison of Situation and Event Calculus. In the following chapters we will concentrate on the Event Calculus, which is sufficiently expressive as long as no counterfactual reasoning is required.

Chapter 6

Knowledge Representation in OLP Event Calculus

In this chapter, we take Event Calculus beyond its usual field of application and show how it can be extended for use in various different knowledge representation settings. In this way we hope to bridge the gap between theoretical approaches to the frame problem in restricted settings, and actual knowledge representation in real-world applications. In section 6.1, we extend the OLP Event Calculus for dealing with continuously changing fluents. Section 6.2 describes how OLP Event Calculus can be used as a general framework for representing temporal knowledge bases. In section 6.3 we apply the formalism in the area of protocol specification. The contributions in this chapter illustrate how OLP Event Calculus meets the central requirement of knowledge representation: using the same representation of a domain in different applications.

6.1 Representing Continuous Change

6.1.1 Introduction

In this section we extend the Event Calculus with a framework for dealing with continuously changing fluents. A rather profound difference with the discrete setting is that the inertia assumption is no longer valid and should be replaced with more complex laws. We will show that this absence of inertia leads to many complications that are absent in the discrete case.

As an example of continuous change we take the problem of a tank that can be filled with water, as introduced in [100]. This is a simple problem, yet it illustrates many of the complications that continuous change gives rise to. The most important complication is the problem of autotermination, also identified in [100]: It is possible, and in fact occurs often, that the change of a fluent over time gets terminated by itself, without the occurrence of any external events. This happens for example when the rising water in a tank reaches the tank's rim. At that moment, the increase of the water level causes the event of its own termination.

Other complications of this form can occur if the water triggers events when it reaches certain levels, like the ringing of a warning bell. In general, a changing fluent can cause any number of events to happen at different time points.

There exist some approaches to dealing with continuous change in temporal reasoning. One is the approach of [94, 95] in the *Features and Fluents* setting of [96] which is based on differential equations. Another approach, introduced in [100] and further used in [80], uses the Event Calculus in a normal logic programming setting. In this approach the continuous fluents are described as exactly known functions of time. Differential equations have also been used in an Event Calculus setting in [74]. In all of these approaches it is assumed that the changing fluent is exactly known as a function of time, or can be calculated from other data like the flow through a certain tap.

We argue that in the case of continuous change — even more than in other cases — it is important to allow for incomplete knowledge. For example, while it is not too hard to check whether a turkey is dead or alive, determining the water level in a filling tank as an exact function of time is non-trivial. Probably the only certainty is that the level is rising. For this reason we adopt a qualitative physics point of view, assuming that the evolution of the continuous fluent is only roughly, qualitatively known (for example we only know it is increasing or decreasing). We will show on a number of examples how this qualitative information can be naturally represented and adequately used for solving the aforementioned types of problems in temporal reasoning.

We present our proposal in two stages: in a first step we deal only with elementary changes, i.e. changes which are the result of one independent cause. In the second step we extend this solution, allowing for any number of simultaneous influences to affect the changing fluent, resulting in arbitrarily complex changes. The solution should fit in with the general framework of the OLP Event Calculus, so that in particular the treatment of discrete fluents is not changed. The various forms of reasoning supported

in discrete Event Calculus should of course also be possible when continuous fluents are involved.

6.1.2 Elementary Changes

We start with the Event Calculus axioms given in Chapter 4. To represent continuous change, we introduce two new predicates *cont_change* and *state_in_change*. The atom $cont_change(P, Sort, T)$ denotes that at time point T the fluent P is subject to a continuous change of sort *Sort*. The following clauses define when a continuous change is in effect.

$$\begin{array}{lll} cont_change(P, Sort, T) & \leftarrow & happens(E_1, T_1), T_1 < T, \\ & & init_change(E_1, P, Sort), \\ & & \neg\, change_clipped(T_1, P, T, Sort). \end{array}$$

$$\begin{array}{lll} change_clipped(T', P, T, Sort) & \leftarrow & happens(E_2, T_2),\ T_1 < T_2,\ T_2 < T, \\ & & term_change(E_2, P, Sort). \end{array}$$

where the predicates *init_change*, *term_change* and *change_clipped* correspond to *initiates*, *terminates* and *clipped* for discrete properties.

Sort is a parameter used to distinguish between different kinds of change, each kind having certain unique properties. How many and which kinds of change are to be distinguished, depends on the amount of available knowledge as well as on the relevance of the observed differences between two kinds. A lot of work on this topic of making useful and adequate abstractions exists in the qualitative physics community (see for example [34] and [60]). One simple and obvious abstraction, which is often used in qualitative physics, is the distinction between positive and negative change. We will use this distinction in our examples.

We assume of course that a fluent can only be subject to one change at any one time, which is represented by the FOL axiom

$$[cont_change(P, Sort, T) \wedge cont_change(P, Sort', T)] \rightarrow Sort = Sort'$$

During periods of change, the predicate $state_in_change/2$ determines the precise value of the changing fluent, as expressed in the following clause:

$$\begin{array}{lll} holds(val(P, X), T) & \leftarrow & cont_change(P, Sort, T), \\ & & state_in_change(P, X, T). \end{array}$$

Note that we use the term $val(P, X)$ to denote "continuous fluent P takes on value X". The use of a *state_in_change* predicate to define the value of a changing fluent is similar to the definition of trajectories in [100],

but where trajectories are defined as known functions of time, we make *state_in_change* an undefined predicate. We use FOL axioms constraining the *state_in_change* predicate to represent the available qualitative knowledge, for example monotonicity or continuity of the change. These axioms will be introduced later on.

When a continuous fluent is not changing, it is subject to the normal inertia axiom of the Event Calculus. The following clauses ensure a correct transition between periods of change and periods of rest:

$$\begin{array}{lcl} terminates(E, val(P, X)) & \leftarrow & init_change(E, P, Sort). \\ initiates(E, val(P, X)) & \leftarrow & event(E, T), holds(val(P, X), T), \\ & & term_change(E, P, Sort), \\ & & \neg init_change(E, P, Sort'). \end{array}$$

i.e. the start of a change terminates a period of rest, while termination of the change initiates a new period of rest.

As we have indicated before, the simple fact that a changing fluent reaches a certain value can trigger a number of effects. In Event Calculus, it is assumed that only events can cause effects (as is obvious in the frame axiom, which contains a *happens* atom). Since due to continuous change effects can be triggered at any time, we say that there is an event associated with each time point (though most of these events have no effects):

$$time(T) \rightarrow \exists E : (happens(E, T))$$

Given that there also is a one-to-one correspondence between events and time points, we find that the distinction between events and time points fades and that it would be possible to retain only one of the concepts. In some formalisations of the Event Calculus this simplification is adopted automatically. To preserve uniformity with the formalisation given before, we choose to distinguish events from time points.

We illustrate the use of this formalism by describing a filling water tank. The constraints we define are problem specific, since they depend on the actual knowledge available about the change. However, most of them represent quite common properties, like continuity of change, and can be generalised or adapted to other problem descriptions.

The changing fluent we consider is *level*, representing the water level in the tank. In this example we choose to distinguish only two kinds of change: rising and dropping water level, denoted by sorts $+$ and $-$. As indicated earlier, this is a very simple abstraction, but we will show its uses. The tank contains a tap and a plug. An open tap results in rising water level,

an open plug in dropping level.

$$\begin{array}{lcl} init_change(E, level, +) & \leftarrow & act(E, open_tap). \\ term_change(E, level, +) & \leftarrow & act(E, close_tap). \\ init_change(E, level, -) & \leftarrow & act(E, open_plug). \\ term_change(E, level, -) & \leftarrow & act(E, close_plug). \end{array}$$

We impose that the tap and the plug cannot be simultaneously open, as in this first step we do not allow multiple influences on one fluent at the same moment.

$$\neg(cont_change(level, -, T) \wedge cont_change(level, +, T))$$

We then need to formalise what we know about this water level, which is the following:

- The water level is rising (dropping) monotonically.
- At any instant in time, the water is at only one level.
- The change is continuous, i.e. if the water reaches two different levels during one period of change, then it will also reach all levels between them.
- If the tap is opened, and nothing happens that stops the rising of the water, then the water will eventually reach the rim of the tank: in other words we assume the water level will not show strange behaviour like converging asymptotically to a finite value. Similarly the tank will eventually become empty if the plug is open.
- When the rising (dropping) water reaches the rim (bottom) of the tank, the change is automatically terminated.

This information is expressed in a number of FOL axioms. In these axioms, we make use of the following concept representing that two time points belong to a same period of change on a certain fluent:

$$\begin{array}{lcl} same_change(P, Sort, T_1, T_2) & \leftarrow & happens(E, T),\ T \leq T_1,\ T_1 < T_2, \\ & & init_change(E, P, Sort), \\ & & \neg change_clipped(T, P, T_2, Sort). \end{array}$$

where $\leq$ is defined in the usual way.

Since we are dealing with continuous fluents, which can take on a wide range of values, we need to provide some information on these possible values. In general we use the atom $isa(X, O)$ to denote that the value

X belongs to the set of values O. We also define a linear order on each set of values, using the predicate $less/3$: $less(O, X, Y)$ denotes that X is strictly less than Y according to the order on O. $less/3$ is an open predicate satisfying

$$\begin{array}{l}
\neg(less(O,X,Y) \wedge less(O,Y,X)) \\
(less(O,X,Y) \wedge less(O,Y,Z)) \rightarrow less(O,X,Z) \\
(isa(X,O) \wedge isa(Y,O)) \rightarrow [less(O,X,Y) \vee less(O,Y,X) \vee (X = Y)] \\
less(O,X,Y) \rightarrow (isa(X,O) \wedge isa(Y,O))
\end{array}$$

Moreover, $min(O)$ and $max(O)$, if they exist, denote the extrema of O:

$$\neg less(O, max(O), X) \wedge \neg less(O, X, min(O)))$$

The set O does not need to be completely determined: isa can in general be left open, like $time$.[1]

We refer to the set of water levels as l_type, and add to our water tank example the constraint

$$holds(val(level, X), T) \rightarrow isa(X, l_type).$$

Now we can formulate our constraints on changes. These FOL axioms are written in terms of *holds*, but actually — through the rules for *holds* in terms of *state_in_change* — constrain this open predicate. We write the constraints in the form of a definition of *invalid* (as explained in Chapter 2) to give a flavour of what they look like in this format. The first of the axioms, the monotonicity axiom, ensures that for every pair of time points during the same period of positive change, the level on the later time point is greater than the level on the earlier one. The opposite holds for negative change.

$$\begin{array}{lll}
invalid & \leftarrow & same_change(level, +, T_1, T_2),\ holds(val(level, X), T_1), \\
 & & holds(val(level, Y), T_2),\ T_1 < T_2,\ less(l_type, Y, X). \\
invalid & \leftarrow & same_change(level, +, T_1, T_2),\ holds(val(level, X), T_1), \\
 & & holds(val(level, X), T_2),\ T_1 < T_2,\ \neg X = max(l_type). \\
invalid & \leftarrow & same_change(level, -, T_1, T_2),\ holds(val(level, X), T_1), \\
 & & holds(val(level, Y), T_2),\ T_1 < T_2,\ less(l_type, X, Y). \\
invalid & \leftarrow & same_change(level, -, T_1, T_2),\ holds(val(level, X), T_1), \\
 & & holds(val(level, X), T_2),\ T_1 < T_2,\ \neg X = bottom(l_type).
\end{array}$$

[1] The order $<$ on time points is a variant of this general linear order, with t_0 being the minimum of the set and no maximum defined. Similarly *time* is a special case of *isa*.

We allow for the level to remain constant once the water reaches the rim. This, together with the restriction that the rim is the maximum existing level, captures the idea of autotermination: when the tank is full, the level stops rising. We choose this representation rather than introducing a terminating event caused by the change like Shanahan does. Such autoterminating event would not distinguish the case in which the tap is closed just when the tank is full from the case in which the tank overflows. However, there are differences (like the floor getting wet). Moreover, in our next step we will allow for multiple simultaneous influences on a changing variable. In that case, the introduction of an autoterminating event leads to erroneous conclusions, as we will discuss later. In our representation the tank being full does not terminate the period of change, even though the level remains constant.
The other constraints look like this:

- no two levels at the same instant:
  $invalid \leftarrow holds(val(level, X), T_1), holds(val(level, Y), T_1), X \neq Y.$

- continuity:
  $invalid \leftarrow same_change(level, Sort, T_1, T_2), T_1 < T_2, holds(val(level, X), T_1), holds(val(level, Y), T_2), isa(Z, l_type), between(l_type, Z, X, Y), \neg reach_in(level, Z, T_1, T_2).$
  $reach_in(level, Z, T_1, T_2) \leftarrow happens(E_3, T_3), T_1 < T_3, T_3 < T_2, holds(val(level, Z), T_3).$
  $between(O, Z, X, Y) \leftarrow less(O, X, Z), less(O, Z, Y).$
  $between(O, Z, X, Y) \leftarrow less(O, Y, Z), less(O, Z, X).$

- water eventually reaches the rim:
  $invalid \leftarrow happens(E, T), init_change(E, level, +), \neg change_clipped_after(T, level, +), isa(max(l_type), l_type), \neg reach_after(T, level, max(l_type)).$
  $invalid \leftarrow happens(E, T), init_change(E, level, -), \neg change_clipped_after(T, level, -), isa(min(l_type), l_type), \neg reach_after(T, level, min(l_type)).$
  $reach_after(T, P, X) \leftarrow happens(E_2, T_2), T < T_2, holds(val(P, X), T_2).$
  $change_clipped_after(T, P, Sort) \leftarrow happens(E_2, T_2), T < T_2, term_change(E_2, P, Sort).$

These constraints conclude the description of the water tank problem domain.

6.1.3 Some Applications

Already in this simplified version of our proposal we can study various kinds of problem solving. We start with a very simple scenario. We assert three relevant levels: the bottom of the tank ($min(l_type)$), the rim ($max(l_type)$), and a level halfway. The tank is initially empty, and then a tap is opened.

For simplicity reasons we define the predicates *isa*, *happens* and *time* by enumeration instead of leaving them open. In this way we avoid obtaining an infinite number of answers from SLDNFA (though from a declarative point of view an infinite number of answers is not problematic, it is rather impractical). In the current scenario, three events are sufficient.

$$\begin{array}{ll} time(t_0). & happens(start, t_0). \\ time(t_1). & happens(e_1, t_1). \\ time(t_2). & happens(e_2, t_2). \\ time(t_3). & happens(e_3, t_3). \\ & \\ isa(min(l_type), l_type). & initially(val(level, min(l_type))). \\ isa(half, l_type). & act(e_1, open_tap). \\ isa(max(l_type), l_type). & \end{array}$$

$$t_0 < t_1 \quad t_1 < t_2 \quad t_2 < t_3$$

A first question about this partial scenario could be "What happens after we open the tap ?" In other words, we want to know if there is a complete scenario consistent with the given theory, and if so which one. This is a temporal projection problem: given an initial state, we compute the evolution of the world from that state on. The above question can be answered by presenting the goal $\leftarrow true$ to SLDNFA and looking at the abduced facts. We obtain the following facts:

$$\begin{array}{l} state_in_change(level, half, t_2). \\ state_in_change(level, max(l_type), t_3). \end{array}$$

and a couple of facts ordering the *l_type* and *time* sets. Both orders are trivially determined by the given constraints.

This solution is indeed the intended one: because the tap is never closed, the water reaches the rim of the tank. This can happen no later than at time t_3, since t_3 is the latest time point we defined. Now, since the water reaches the rim, we know it also reaches all levels between bottom and rim. So, the water must reach the level halfway at some time between t_1 and t_3, which can only be at t_2, the only other time point we provided. If we had asserted that other time points existed or could exist between t_1 and t_3 or

after t_3, which we should allow in general, there would be other solutions as well.

In a second example we add an alarm bell that rings when the water reaches the level halfway. The example shows how events that are caused by changing fluents can be handled. At the same time we add an aspect of uncertainty, by saying that possibly the alarm bell is broken. The effect on the bell is represented by the clause

$$\begin{array}{lcl} initiates(E, ring_bell) & \leftarrow & happens(E,T),\ holds(val(level, half), T), \\ & & \neg broken_bell. \end{array}$$

and as we have no information on the state of the bell, *broken_bell* is an open predicate. With the bell and the uncertainty added, we can demonstrate how diagnosis problems are handled. Assume we have the same scenario as above, and in addition we observe that the bell is not ringing in the end (i.e. at t_3). We want to know how this is possible, to which end we can present the query

$$\leftarrow \neg holds(ring_bell, t_3).$$

We find the same set of abduced facts as above, plus the additional fact *broken_bell*: the water behaves just like before, so at t_2 the level halfway is reached, at which point the bell should start ringing unless it is broken. A solution without *broken_bell* does not exist.

Finally we can use the SLDNFA procedure on this representation for planning: we generate a sequence of actions that, after we have opened a tap, leads to an end state in which the bell is not ringing even though it is not broken. We have the general rules

$$\begin{array}{lcl} init_change(E, level, +) & \leftarrow & act(E, open_tap). \\ term_change(E, level, +) & \leftarrow & act(E, close_tap). \\ initiates(E, ring_bell) & \leftarrow & holds(val(level, half), E),\ \neg broken_bell. \end{array}$$

The definitions of *time*, *happens*, $<$, *initially* and *isa* are identical to those in our first application (so again we simplify the problem by stating there are three events), and *state_in_change*, *less*, *act* and *broken_bell* are open. There is one constraint on *act*:

$$act(e_1, open_tap)$$

If we solve the query

$$\leftarrow t_0 < t_3,\ \neg holds(ring_bell, t_3),\ \neg broken_bell.$$

we find a solution with abduced facts

$$act(e_1, open_tap).$$
$$act(e_2, close_tap).$$

and a second one with the *close_tap* action on e_3 instead of e_2. Because the tap is now closed at a certain point in time, the water is no longer guaranteed to reach the rim, or even the level halfway (this depends on the flow through the tap). The level at time t_3 will be somewhere between $min(l_type)$ and *half*.

Note that especially in planning one should be very careful when asserting the occurrence of events. In general *happens*, like *act*, should be an open predicate, and it is well possible that by asserting too few events one will not obtain the intended solutions. On the other hand, leaving *happens* open can cause SLDNFA to go into a loop in which it keeps generating useless events. This is of course a problem of the procedure and not of the theory. In practice the problem is solved by employing an iterative deepening search on events: first answers with one event are considered, then answers with two events, and so on. In the examples in this section, the answers we have obtained by precisely enumerating the events are those with a minimal number of events, i.e. those that are obtained first.

6.1.4 Changes Caused by Multiple Influences

If we want to allow for multiple simultaneous influences on the same continuous fluent, we need to extend our proposal: in the water tank example, up to now we could immediately link an open tap or plug to a change of a particular type. When multiple taps and plugs are present which can be open at the same time, we must distinguish between influences on the changing fluent and the change itself. Hence we introduce the concept of an *influence*, represented by a new predicate *influence*/4. $influence(I, P, Sort, T)$ holds if at time point T, P is subject to the influence I of sort *Sort*.

Changes are now defined in terms of the existing influences, while the effect of actions is the initiation and/or termination of these influences. The predicates *init_change*, *term_change* and *change_clipped* no longer have any meaning and they are replaced with a set of new predicates *init_influ*, *term_influ*, *influ_clipped*, *influ_started* and *influ_change*. The following

new definitions apply:

$$
\begin{array}{lcl}
influence(I,P,S,T) & \leftarrow & happens(E_1,T_1),\ T_1 < T, \\
 & & init_influ(E,I,P,S), \\
 & & \neg influ_clipped(I,T_1,P,T). \\
 & & \\
influ_clipped(I,T_1,P,T) & \leftarrow & happens(E_2,T_2),\ T_1 < T_2,\ T_2 < T_1, \\
 & & term_influ(E_2,I,P,S). \\
influ_started(I,T_1,P,T) & \leftarrow & happens(E_2,T_2),\ T_1 < T_2,\ T_2 < T_1, \\
 & & init_influ(E_2,I,P,S). \\
 & & \\
influ_changed(T_1,P,T_2) & \leftarrow & influ_clipped(I,T_1,P,T_2). \\
influ_changed(T_1,P,T_2) & \leftarrow & influ_started(I,T_1,P,T_2).
\end{array}
$$

We impose a consistency condition stating that an influence cannot at the same time be initiated and terminated:

$$\neg(init_influ(T,I,P,S) \wedge term_influ(T,I,P,S))$$

We redefine *cont_change* in terms of influences, and leave our frame axioms and constraints unchanged. We choose to distinguish two kinds of influence: positive and negative. This leads to three kinds of change: if all influences on a variable are positive, the change is positive (+). If all influences are negative, the change is negative (−). If there are both positive and negative influences, the change is continuous but with unknown (and possible varying) direction (?). This is expressed in the following rules:

$$
\begin{array}{lcl}
cont_change(P,+,T) & \leftarrow & influence(I,P,+,T),\ \neg any_influ(P,-,T). \\
cont_change(P,-,T) & \leftarrow & influence(I,P,-,T),\ \neg any_influ(P,+,T). \\
cont_change(P,?,T) & \leftarrow & influence(I,P,+,T),\ influence(J,P,-,T). \\
 & & \\
any_influ(P,Sort,T) & \leftarrow & influence(J,P,Sort,T).
\end{array}
$$

As in the first proposal, the types of change and influence are chosen because of their generality. They can be modified if the problem domain allows or requires this, for example when a distinction can be made between slow change and fast change. We choose to stick with this very general set of types, which is already sufficiently detailed to illustrate our proposal.

The correct transition between periods of rest and periods of change is

ensured by the following rules:

$$\begin{array}{lcl} terminates(E, val(P, X)) & \leftarrow & init_influ(E, I, P, S). \\ initiates(E, val(P, X)) & \leftarrow & happens(E, T),\ holds(val(P, X), T), \\ & & term_influ(E, I, P, S), \\ & & \neg persisting_inf(P, T). \end{array}$$

$$\begin{array}{lcl} persisting_inf(P, T) & \leftarrow & happens(E, T),\ init_influ(E, I, P, S). \\ persisting_inf(P, T) & \leftarrow & happens(E^*, T^*),\ influence(I, E^*, P, S), \\ & & \neg term_influ(E, I, P, Sort). \end{array}$$

where $persisting_inf(P, T)$ denotes that there are influences on P that will continue to exist after T (or start to exist at T).

We can now model the water tank in the following way, adding the possibility of multiple taps and plugs:

$$\begin{array}{lcl} init_influ(E, tap(Y), level, +)) & \leftarrow & act(E, open_tap(Y)). \\ init_influ(E, plug(Y), level, -)) & \leftarrow & act(E, open_plug(Y)). \\ term_influ(E, tap(Y), level, +)) & \leftarrow & act(E, close_tap(Y)). \\ term_influ(E, plug(Y), level, -)) & \leftarrow & act(E, close_plug(Y)). \end{array}$$

using the name or number of the tap to identify the influence. In this way, it is easy to determine which influence is initiated or terminated by an action.

The monotonicity, continuity and unique level constraints do not need to be modified. However, the constraints indicating that the water eventually reaches the rim or the bottom — when rising or dropping — get more complicated because of the possibility of many influences: if, at any point in time, there are persisting positive influences while no negative influences remain, and if after that time point there is no change of influence anymore, then the water will eventually reach its maximum level. Again a similar conclusion holds for the water reaching its minimum level if only negative influences persist.

$$\begin{array}{lcl} invalid & \leftarrow & happens(E, T),\ persisting_inf(T, level, +), \\ & & \neg persisting_inf(T, level, -), \\ & & \neg influ_change_after(T, level), \\ & & isa(max(l_type), l_type),\ \neg reach_after(T, max(l_type)). \end{array}$$

$$\begin{array}{lcl} invalid & \leftarrow & happens(E, T)),\ persisting_inf(T, level, -), \\ & & \neg persisting_inf(T, level, +), \\ & & \neg influ_change_after(T, level),\ isa(min(l_type), l_type), \\ & & \neg reach_after(T, min(l_type)). \end{array}$$

with *reach_after* as before, and *influ_change_after* defined as

$$\begin{array}{lcl} influ_change_after(T,P) & \leftarrow & happens(E_2,T_2),\ T < T_2, \\ & & term_influ(E_2,I,P,S). \\ influ_change_after(T,P) & \leftarrow & happens(E_2,T_2),\ T < T_2, \\ & & init_influ(E_2,I,P,S). \end{array}$$

This completes our proposal for continuous change with multiple influences. Basically it can handle the same kinds of problems as our first proposal, but it eliminates the unrealistic restriction to one influence. An example that shows how even changes with unknown direction contain possibly valuable information, is the following scenario: we open a tap and a plug, resulting in an unknown change. We know nothing about the initial water level. We observe the water level at t_1 and t_2, and see it is below halfway at t_1 and above halfway at t_2. In this case, the bell should be ringing at t_2 if it is not broken. Apart from the general rules given above, we have

$$\begin{array}{lcl} initiates(E, ring_bell) & \leftarrow & happens(E,T),\ holds(val(level,half),E), \\ & & \neg broken_bell. \end{array}$$

$$\begin{array}{ll} isa(min(l_type), l_type). & isa(x, l_type). \\ isa(max(l_type), l_type). & isa(y, l_type). \\ isa(half, l_type). & \end{array}$$

$$\begin{array}{ll} happens(start, t_0). & t_0 < t_1 \\ happens(e_1, t_1). & t_1 < t' \\ happens(e', t'). & t' < t_2 \\ happens(e_2, t_2). & t_2 < t_3 \\ happens(e_3, t_3). & \end{array}$$

$$act(e_1, open_tap(tap_1)). \qquad act(e_1, open_plug(plug_1)).$$

We add an extra event e' between t_1 and t_2. If happens were open, this event would always be abduced, as the constraints can never be satisfied otherwise. For the reasons indicated earlier we choose to simply add this necessary event to the scenario. Of course we do not give any information about what is going on at time t'. Moreover we add two new levels x and y, which are the levels we observe. The observations, written as FOL axioms, are

$$\begin{array}{ll} holds(val(level,x),t_1) & less(l_type, x, half) \\ holds(val(level,y),t_2) & less(l_type, half, y) \end{array}$$

The query then is

$$\leftarrow \neg holds(ring_bell, t_2).$$

and again, we find that *broken_bell* is abduced. Indeed, the change is not required to be monotonic, and we do not know how the water level behaves between t_1 and t_2, whether it reaches the rim or the bottom, or how many times it passes the level halfway. Yet we do know, because of continuity, that it passes the level halfway at least once between t_1 and t_2. Therefore, if the bell is not ringing, it has to be broken.

We conclude this section by indicating why introducing autotermination events would lead to erroneous results in the extended version of our proposal, as we claimed above. Suppose we say an autotermination event occurs if the water reaches the rim of the tank. If we open a tap and wait long enough, such an event will eventually occur. Suppose then subsequently we do not close the tap, but open a plug. We could then conclude that there is a negative influence from the plug, but no positive one from the tap, since that influence was (auto)terminated by the event. Therefore we could conclude that the tank would empty. However, in reality the positive influence of the tap still exists. Though it has no more effects if it is the only influence present, it is not terminated and can still show itself by counteracting or interacting with other influences.

This does not mean, however, that autotermination events are a worthless notion: they can indeed occur in reality. As an example, there could be a sensor at the rim of the tank that detects the water level reaching it. This could provoke the closing of all taps. In that case, we have a real autotermination event (which is similar to the events triggering alarm bells when the water reaches certain levels), and it has to be represented as such.

6.1.5 Discussion

We have extended the OLP Event Calculus with a representation of continuous change, assuming that we have only qualitative knowledge on that change. We have used an open predicate to represent the unknown value of the continuous fluent during periods of change, and added FOL axioms representing the available partial knowledge. We have made a distinction between the influences on a changing fluent and the change itself, and illustrated that this distinction is necessary if we want to model any but the most simple problem domains.

A few other authors have addressed the problem of representing continuous change in a temporal reasoning formalism. Allen's theory of time ([3]) was modified in [35] to fix certain problems arising when continuous change was considered in the original theory. Sandewall ([94], [95]) describes a framework that uses differential equations combined with first order logic and a form of chronological minimisation of change.

In an approach based like ours on the Event Calculus, Shanahan ([100]) extends the formalism with *trajectories*. These trajectories describe periods of continuous change in which the fluent value is exactly known as a function of time. The extension fits in nicely with the Event Calculus, as periods of rest — described by the basic Event Calculus axioms — and periods of motion — described by the axioms for trajectories — interact without a problem. This solution is further refined in [80] to make reasoning at different levels of time granularity possible, and to allow for the parameters of the change to be modified while the change is in effect.

This approach assumes that each trajectory is exactly known. To avoid this, in [101] a qualitative version of trajectories is proposed, based on the naive physics theory of confluences described in [22]. Confluences are, simply stated, a form of qualitative differential equations. They can be used to describe the world in terms of the signs of certain quantities and the signs of their derivatives, without knowing any exact values. Shanahan combines these confluences with trajectories. The new trajectories do not need to be exactly known, but are qualitative.

The above description indicates that Shanahan's approach shows similarities to ours. We indicate some of the more important differences. One difference is that in Shanahan's approach a number of *landmark values* are distinguished through which the value of a changing fluent can pass. This corresponds to our definition of levels. However, where in our proposal the set of levels or the order on it can be incompletely specified, the set of landmark values is fixed. Additional fluents $between(X, A, B)$, meaning that the variable X is between the landmark values A and B, are introduced to deal with intermediate levels. In our approach this is accomplished by assuming a new unknown level L between A and B and stating that X is at L.

A second difference is that to ensure monotonicity, continuity and similar conditions Shanahan uses a set of rules involving skolem functions that determine the value of the changing quantity at T in terms of the value just before T. In our approach this is achieved by explicitly imposing constraints.

A third difference lies in Shanahan's treatment of autotermination using a caused event, which we have avoided due to the possible problems indicated earlier. Finally, Shanahan's proposal does not distinguish influences from changes, a distinction which is necessary for dealing with simultaneous influences on the same variable.

If we want to use our representation in other problem domains than the water tank world, we have to substitute whatever changing quantity we describe for *level* in the constraints. In other words, we have to define the

constraints separately for each changing variable, just like Shanahan's trajectories are defined independently for each change. Of course each change may satisfy a different set of constraints, since the available knowledge can vary a lot.

The types of change we defined for the water tank problem (rising, dropping, unknown) are very general and can be used in many applications. However, more types can be distinguished if an application requires this, and the constraints can be modified depending on the available information.

We have shown how SLDNFA supports various types of problem solving starting from our representation. This is another aspect in which the approach differs from existing ones, since as far as we know all representations of continuous change to date, be it qualitative or quantitative, only support solving temporal projection problems (if problem solving is at all supported). Meanwhile, we hope our examples have shed some light on how problem solving in open logic programming is supported by a procedure like SLDNFA.

6.2 A Framework for Temporal Knowledge Bases

6.2.1 Introduction

In this section we use the OLP Event Calculus as a general framework for representing temporal knowledge bases. We illustrate that open logic programming is a very suitable language for representing a knowledge base containing incomplete information, and how the Event Calculus deals correctly with the temporal aspects. Moreover we show how the basic functionality of a knowledge base can in principle be implemented using the SLDNFA procedure, and how in addition we could use this procedure for tasks essential to the manipulation of the knowledge base as well as for general problem solving using the data, without resorting to a different representation. This illustrates the proclaimed advantage of declarative knowledge representation in practice: the same representation can be used for different purposes at several levels.

Kowalski ([58]) has argued earlier that the Event Calculus in Logic Programming can be used to formalise the evolution of a database system. The work presented here addresses different issues than Kowalski's work, in particular the representation of incomplete information in open logic programming and the use of an abductive procedure for implementing the

functionality of the knowledge base.

The precise aims of this work (the desired types of formulae in the knowledge base, the topology of time, and the knowledge base's functionality, as described below) have been inspired by the project proposal described in [39]: the proposed project had as goal the building a temporal knowledge base which could easily be used by an intelligent agent for various kinds of problem solving. In particular this implied that the knowledge base representation had to be as close as possible to the representation used by applications using the knowledge base. The project has not been carried out, but we intend to show how its main goals can be accomplished by OLP Event Calculus and SLDNFA.

In section 6.2.2 we specify the desired contents and functionality of the knowledge base. Section 6.2.3 presents the proposed representation, and in section 6.2.4 we discuss how to provide the basic functionality, followed by a detailed example in section 6.2.5. More advanced functionality which can be implemented using SLDNFA and problem solving using the knowledge base are discussed in section 6.2.6, and in section 6.2.7 we conclude with a discussion.

6.2.2 Specification of the Knowledge Base

Topology of time

When representing time in a knowledge base, we have a choice of several topologies, as described in [39]. We can consider time to be an ordered set of points, with no calculus defined on them, in which the only allowed expressions are formulae $t = t'$ and $t < t'$. Another choice is a numerical time line, where operations like addition are possible. In that case we can have expressions like $t_1 - t_2 < 3$. Finally, time can be seen as a set of intervals, with the thirteen relations proposed in [3] as possible relations between each pair of intervals. We have chosen to combine the first and third options, in other words allowing for both time points and intervals in the knowledge base, but not for numerical constraints. We relate time points and intervals by considering intervals as periods of time started and ended by a time point. Evidently in the context of a knowledge base time is linear, not branching.

Contents of the knowledge base

The data we aim to represent in the knowledge base are formulae representing the truth value of fluents during intervals and at time points, and the change of truth values at certain time points. Further, formulae repre-

senting the order on time points and the relations between intervals will be used. We propose the following set of basic formulae, with P a fluent:

- $holds_at(P, T)$: P is true at time point T.
- $holds_in(P, int(T_1, T_2))$: P is true throughout the interval $int(T_1, T_2)$. This interval does not need to be "maximal": P may remain true after T_2 or can be true already before T_1.
- $notholds_in(P, int(T_1, T_2))$: P is false throughout $int(T_1, T_2)$.
- $on(P, T)$: P's value changes from false to true at time point T.
- $off(P, T)$: P changes from true to false at T.

The possible relations between time points and intervals are represented by the following formulae. (In the case of intervals, we distinguish thirteen possible relations, based on those defined in [3], though some names may differ.)

- $T_1 = T_2$: T_1 and T_2 are the same time point.
- $T_1 < T_2$: T_1 is chronologically before T_2.
- $equal(I_1, I_2)$: I_1 and I_2 are the same interval.
- $meets(I_1, I_2)$: the end point of I_1 is the starting point of I_2.
- $overlaps(I_1, I_2)$: I_1 starts before I_2, and ends during I_2.
- $starts(I_1, I_2)$: I_1 is an initial subinterval of I_2.
- $ends(I_1, I_2)$: I_1 is a terminal subinterval of I_2.
- $during(I_1, I_2)$: I_1 is a subinterval of I_2 that is initial nor terminal.
- $before(I_1, I_2)$: the end of I_1 lies strictly before the start of I_2.
- $after(I_1, I_2)$: $before(I_2, I_1)$.
- $metby(I_1, I_2)$: $meets(I_2, I_1)$.
- $overlapby(I_1, I_2)$: $overlaps(I_2, I_1)$.
- $startby(I_1, I_2)$: $starts(I_2, I_1)$.
- $endby(I_1, I_2)$: $ends(I_2, I_1)$.
- $contains(I_1, I_2)$: $during(I_2, I_1)$.

We can build more complex expressions by combining these basic formulae using the classical logical connectives and quantifiers : if P and Q are valid expressions, then $(\neg P)$, $(P \wedge Q)$, $(P \vee Q)$, $(P \oplus Q)$ (exclusive or), $(P \rightarrow Q)$, and $(P \leftrightarrow Q)$ are valid as well, and if P is an expression, then so are $(\forall X : P)$ and $(\exists X : P)$.

Desired functionality

The formulae defined above determine the possible contents of the temporal knowledge base. The functionality we require of such a knowledge base is the following:

- checking whether a knowledge base KB is consistent.
- answering simple conjunctive queries as well as more complex ones. Since the data may be incomplete, we must distinguish between two types of query:
  1. "Is Q necessarily true in KB ?" (does $KB \models Q$?)
  2. "Is Q possible in KB ?" (is $Q \cup KB$ consistent ?).
- in the case of inconsistency, proposing solutions to restore consistency.
- finally, and maybe most importantly, using the knowledge base for problem solving in temporal domains, in particular planning.

6.2.3 Representing the Data in OLP Event Calculus

In the representation of the knowledge base, we found it essential to split the theory in two parts: basically an A-Box and a T-Box, as discussed in Chapter 3. The T-Box is a set of logic programming clauses defining all basic formulae used in the knowledge base in terms of primitive Event Calculus predicates. This part is independent of the actual contents of the knowledge base. The A-Box contains the actual data in the knowledge base: these are considered to be a set of FOL axioms constraining the possible states of the knowledge base.

As primitive predicates we choose *happens*, $<$, *initiates* and *terminates*. The basic formulae of the knowledge base are defined in terms of these four open predicates, by the following T-Box clauses:[2]

$$\begin{array}{lll}
holds_at(P,T) & \leftarrow & happens(E_1,T_1),\ initiates(E_1,P), \\
 & & T_1 < T,\ \neg clipped(T_1,P,T). \\
holds_in(P,int(T_1,T_2)) & \leftarrow & interval(T_1,T_2),\ holds_from(P,T_1), \\
 & & \neg clipped(T_1,P,T_2). \\
notholds_in(P,int(T_1,T_2)) & \leftarrow & interval(T_1,T_2), \\
 & & notholds_from(P,T_1), \\
 & & \neg started(T_1,P,T_2). \\
 & & \\
interval(T_1,T_2) & \leftarrow & happens(E_1,T_1), T_1 < T_2, \\
 & & happens(E_2,T_2).
\end{array}$$

$$\begin{array}{lll}
started(T',P,T) & \leftarrow & happens(E'',T''), initiates(E'',P), \\
 & & (T' < T''), (T'' < T). \\
clipped(T',P,T) & \leftarrow & happens(E'',T''), terminates(E'',P), \\
 & & (T' < T''), (T'' < T).
\end{array}$$

[2]Note that the difference between events and time points is also in this setting entirely immaterial. We preserve the distinction for clarity reasons.

$$
\begin{array}{lll}
holds_from(P,T) & \leftarrow & happens(E,T),\ initiates(E,P). \\
holds_from(P,T) & \leftarrow & happens(E,T),\ holds_at(P,T), \\
 & & \neg terminates(E,P). \\
 & & \\
notholds_from(P,T) & \leftarrow & happens(E,T),\ terminates(E,P). \\
notholds_from(P,T) & \leftarrow & happens(E,T),\ \neg holds_at(P,T), \\
 & & \neg initiates(E,P). \\
 & & \\
on(P,T) & \leftarrow & happens(E,T),\ initiates(E,P), \\
 & & \neg holds_at(P,T). \\
off(P,T) & \leftarrow & happens(E,T),\ holds_at(P,T), \\
 & & terminates(E,P).
\end{array}
$$

The definition of *holds_at* is exactly the Event Calculus frame axiom. It follows from our definitions that the interval $int(t_1, t_2)$ actually denotes the interval $]t_1, t_2]$, containing its end point but not its starting point. We choose half-open intervals because working with closed intervals can lead to inconsistencies (one time point can belong to two intervals with different values for the same property), while open intervals lead to time points where properties are undefined. A choice between the two types of half-open intervals is easy: the definition of the Event Calculus naturally leads to the form $]t_1, t_2]$.

The chronological relations between intervals are expressed in terms of relations between their starting points and end points.

$$
\begin{array}{lll}
equal(int(T_1,T_2),int(T_1,T_2)) & \leftarrow & interval(T_1,T_2). \\
meets(int(T_1,T_2),int(T_2,T_3)) & \leftarrow & interval(T_1,T_2), interval(T_2,T_3). \\
overlaps(int(T_1,T_2),int(T_3,T_4)) & \leftarrow & interval(T_1,T_2), interval(T_3,T_4), \\
 & & T_1 < T_3,\ T_3 < T_2,\ T_2 < T_4. \\
starts(int(T_1,T_2),int(T_1,T_3)) & \leftarrow & interval(T_1,T_2), interval(T_1,T_3), \\
 & & T_2 < T_3. \\
ends(int(T_1,T_2),int(T_3,T_2)) & \leftarrow & interval(T_1,T_2), interval(T_3,T_2), \\
 & & T_3 < T_1. \\
during(int(T_1,T_2),int(T_3,T_4)) & \leftarrow & interval(T_1,T_2), interval(T_3,T_4), \\
 & & T_3 < T_1,\ T_2 < T_4. \\
before(int(T_1,T_2),int(T_3,T_4)) & \leftarrow & interval(T_1,T_2), interval(T_3,T_4), \\
 & & T_2 < T_3.
\end{array}
$$

$$
\begin{array}{lll}
after(int(T_1, T_2), int(T_3, T_4)) & \leftarrow & interval(T_1, T_2), interval(T_3, T_4), \\
 & & T_4 < T_1. \\
overl_by(int(T_1, T_2), int(T_3, T_4)) & \leftarrow & interval(T_1, T_2), interval(T_3, T_4), \\
 & & T_3 < T_1,\ T_1 < T_4,\ T_4 < T_2. \\
metby(int(T_1, T_2), int(T_3, T_1)) & \leftarrow & interval(T_1, T_2), interval(T_3, T_1). \\
startby(int(T_1, T_2), int(T_1, T_3)) & \leftarrow & interval(T_1, T_2), interval(T_1, T_3), \\
 & & T_3 < T_2. \\
endby(int(T_1, T_2), int(T_3, T_2)) & \leftarrow & interval(T_1, T_2), interval(T_3, T_2), \\
 & & T_1 < T_3. \\
contains(int(T_1, T_2), int(T_3, T_4)) & \leftarrow & interval(T_1, T_2), interval(T_3, T_4), \\
 & & T_1 < T_3,\ T_4 < T_2.
\end{array}
$$

To this set of definitions, we need to add a couple of general constraints on temporal domains, like in the previous chapters: in particular we impose the consistency condition on initiation and termination and the linear time constraints.

$$\neg(initiates(E, P) \wedge terminates(E, P))$$

$$
\begin{array}{c}
\neg((T_1 < T_2) \wedge (T_2 < T_1)) \\
((T_1 < T_2) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_3) \\
(time(T_1) \wedge time(T_2)) \rightarrow [(T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)] \\
(T_1 < T_2) \rightarrow (time(T_1) \ \wedge \ time(T_2))
\end{array}
$$

Note that these general constraints are FOL axioms, but nevertheless belong to the T-Box. They are similar to the general constraints involving constructs like "$\sqsubseteq$" in description logics, which belong to the T-Box without being concept definitions.

The open logic program defined so far determines the terminology of our knowledge base. The actual data reside in the A-Box as a set of FOL axioms. These can be basic formulae as well as complex expressions. Some examples:

$$
\begin{array}{c}
notholds_in(has(john, book_1), int(t_1, t_2))) \\
holds_at(p(a), T) \rightarrow holds_at(q(b), T) \\
\forall T : (holds_at(p, T)) \\
meets(int(t_1, t_2), int(t_3, t_4)) \oplus metby(int(t_1, t_2), int(t_3, t_4)) \\
holds_at(has(X, O), T) \wedge holds_at(has(Y, O), T) \rightarrow X = Y
\end{array}
$$

As usual, we assume free variables to be universally quantified with maximal scope. From a representational point of view, complex data are not problematic. However they will require special attention in the procedure.

The data in the knowledge base are in general incomplete, so possibly many different models exist. For example, consider the knowledge base containing only two simple observations:

$$holds_in(has(john, book_1), int(t_1, t_2)).$$
$$notholds_in(has(mary, book_2), int(t_3, t_4)).$$

We do not know anything about John having the book outside of the interval $int(t_1, t_2)$. He can own it all the time, or only during the mentioned time period, or during a period that starts at t_1 but continues after t_2, and so on. Likewise for Mary's book we have many possible models. Finally we have no information at all concerning the temporal relation linking $int(t_1, t_2)$ and $int(t_3, t_4)$. These periods can overlap, be disjoint, be equal, etc..

We assume the semantics of the proposed theory to be given by the completion semantics for open logic programs of [20] (see Chapter 2 of this thesis), with one exception: we do not include Free Equality axioms for time points, so time constants are treated as skolem constants. In other words, different terms can denote the same time point. This is necessary in the given setting as the relation between different time points, including their possible equality, is in general unknown.

6.2.4 Basic Functionality

We now turn to the issue of providing the required functionality of the knowledge base using SLDNFA. We first study the special case where only basic formulae are allowed in the A-Box.

A theoretical solution

A basic task, which will return in all of the functions we provide, is the generation of a model (in terms of an interpretation for the primitive predicates) of the data. Such a model can easily be found using an abductive procedure like SLDNFA, in one of two equivalent ways: either by trying to solve the goal $\neg invalid$ given the open logic program (which requires writing all the data as a definition of $invalid$, as explained in Chapter 2 and appendix A), or by taking only the T-Box as an open logic program, and trying to solve the query KB (the goal $\leftarrow KB$) obtained as the conjunction of all the data. We adopt the latter approach in this section. SLDNFA returns a set of abduced atoms Δ and a substitution θ such that

$$TBox \cup comp(\Delta) \models KB\theta.$$

As the T-Box contains a set of definitions which uniquely defines all predicates in terms of the open predicates, the set Δ uniquely determines a model of $KB\theta$. Each answer to the query (up to variable renaming) corresponds to one model of the knowledge base, and vice versa.

Checking the consistency of a knowledge base is then straightforward: we check whether KB is consistent by attempting to find an answer to the query KB. The knowledge base is consistent if we obtain an answer, inconsistent if we find failure.

Consider again the knowledge base containing the constraints

$$
\begin{array}{l}
holds_in(has(john, book_1), int(t_1, t_2)). \\
notholds_in(has(mary, book_2), int(t_3, t_4)).
\end{array}
$$

To check its consistency, we form the goal

$$
\begin{array}{ll}
\leftarrow & holds_in(has(john, book_1), int(t_1, t_2)), \\
& notholds_in(has(mary, book_2), int(t_3, t_4)).
\end{array}
$$

We then find for example the abduced facts

$$
\begin{array}{ll}
happens(e_1, t_1) & happens(e_2, t_2) \\
happens(e_3, t_3) & happens(e_4, t_4) \\
t_1 < t_2 & t_2 < t_3 \\
t_3 < t_4 & initiates(t_1, has(john, book_1))
\end{array}
$$

which proves consistency of the data.

Answering queries can be done in a similar way. If we want to know whether Q is possible in the knowledge base KB, we try to abduce a solution that (given the T-Box) entails $KB \wedge Q$. For example, using the same data as above, the query "Is it possible that Mary owns $book_2$ at t_1 ?" will be solved by attempting to solve the goal

$$
\begin{array}{ll}
\leftarrow & holds_in(has(john, book_1), int(t_1, t_2)), \\
& notholds_in(has(mary, book_2), int(t_3, t_4)), \\
& holds_at(has(mary, book_2), t_1).
\end{array}
$$

which has as a model for example

$$
\begin{array}{ll}
happens(e_{new}, t_{new}) & happens(e_1, t_1) \\
happens(e_2, t_2) & happens(e_3, t_3) \\
happens(e_4, t_4) & \\
t_{new} < t_1 & t_1 < t_2 \\
t_2 < t_3 & t_3 < t_4 \\
initiates(e_1, has(john, book_1)) & \\
initiates(e_{new}, has(mary, book_2)). & \\
terminates(e_3, has(mary, book_2)). &
\end{array}
$$

The answer to the query is therefore affirmative.

If the question is whether Q is *necessarily* true given KB, we try to abduce a model for $KB \wedge \neg Q$. If we find no model ($T \wedge KB \wedge \neg Q$ is inconsistent), it follows that $T \wedge KB \models Q$, which is what we were trying to find out. Using the same query as in the previous example, we end up trying to solve

$$\begin{aligned} \leftarrow \quad & holds_in(has(john, book_1), int(t_1, t_2)), \\ & notholds_in(has(mary, book_2), int(t_3, t_4)), \\ & \neg holds_at(has(mary, book_2), t_1). \end{aligned}$$

which has the solution

$$\begin{aligned} & t_1 < t_2 \ \wedge \ t_2 < t_3 \ \wedge \ t_3 < t_4 \\ & initiates(e_1, has(john, book_1)). \end{aligned}$$

so we can conclude that Mary does not necessarily own $book_2$ at t_1.

The above functionality is also an important help when updating the knowledge base: supposedly we would like to ensure that each update leaves the knowledge base in a consistent state, hence a consistency check is required with each update. If inconsistency is detected, the update should be rejected. Alternatively, a warning could be given and the update only allowed after explicit user intervention. Then for example techniques for restoring consistency, like those described below, can be applied.

Practical complications

The above functionality is straightforwardly provided by the existing SLDNFA procedure, except for the treatment of time constants and that of complex data.

First we study the problem of complex data. Just like in Chapter 3 we can use the method of [68] to deal with these. Recall that this transformation transforms general clauses, i.e. clauses of the form

$$A \leftarrow W.$$

with A an atom and W an arbitrary FOL formula, into sets of normal clauses. This is achieved by using a set of rewriting rules, included in appendix A. A goal $\leftarrow W$ can be transformed in the same way, as also indicated in [68]:

Replace	$\leftarrow W.$
by	$\leftarrow answer(X_1 \ldots X_n).$
and	$answer(X_1 \ldots, X_n) \leftarrow W.$

where $X_1 \ldots, X_n$ are the free variables in W.

The resulting rule $answer(X_1 \ldots X_n) \leftarrow W.$ must be transformed further using the rewriting rules for general clauses. The whole transformation is easily automated.

A slightly harder problem is that of time constants: SLDNFA considers the Free Equality axioms to hold for all constants in the data, but in the current setting time constants are intended to be skolem constants, i.e. they are allowed to be equal to one another. We can solve this problem in the following way: we collect all data $(F_1, F_2, \ldots, F_N)$ in the conjunction

$$F_1 \wedge F_2 \wedge \ldots \wedge F_N.$$

We write that conjunction in the form $F(t_1, \ldots, t_n)$ where the t_i are our time point skolem constants. In short, we call this expression F. We can then deskolemise F, i.e. we replace all skolem constants with existentially quantified variables, which results in F':

$$\exists T_1, \ldots, T_n : F(T_1, \ldots, T_n).$$

Skolem's theorem states that for all P and F, with F' the deskolemisation of F as defined above: $P \wedge F$ is consistent if and only if $P \wedge F'$ is consistent. Therefore, replacing skolem constants by existentially quantified variables does not change the result of a consistency check or a query.

We now do the following: before calling the SLDNFA procedure, we build a table linking every time constant to a variable. In the data we pass to SLDNFA, we replace every constant by its corresponding variable. As indicated, this does not change the consistency results.

To find the actual model corresponding to a set Δ produced by SLDNFA, we combine the answer of the SLDNFA procedure with the table of time constants, where some of the variables may be unified by now. In that case the time constants corresponding to these variables are equal in the solution, which is indicated in the answer. We will discuss a detailed example later.

At this point a small word on decidability is required. Normally, an unlimited number of events causes undecidability in OLP Event Calculus: if there is no model of the data, SLDNFA will never terminate but will keep on generating interpretations with more and more events. Luckily, we can limit the search space to solutions with a bounded number of events in a consistency/inconsistency preserving way, thanks to the following result:

Proposition 6.2.1 *If the data in a knowledge base KB are consistent and mention only N different time points, then there exists at least one model of KB containing a number of events* $n \leq 2N$.

Proof:
The proposition is true because of the following: between every two consecutive time points, one event can be assumed to take care of the necessary initiations and terminations of all of the fluents that change value between these points. For each model where two events are inserted between two consecutive time points, there exists a corresponding model where these two events are contracted to one, with the same effect overall effect on all fluents.

In addition, every time point in the data corresponds to an event itself: this is necessary for example when data items $holds_in(p, int(t_1, t_2))$ and $notholds_in(p, int(t_2, t_3))$ are given: t_2 must terminate p, and therefore needs to be an event. Finally we also need an event before the first time point to take care of the first initiations.

This results in $2N$ needed events: one for each time point, one between every two time points and one in the beginning. □

By counting the time points occurring in the data, the program can determine a safe upper bound for the number of events, ensuring that a solution is found if one exists. Note that in the case of complex formulae with existential quantifiers, the existentially quantified time variables are counted as well. In some cases (e.g. when an existential quantifier occurs inside the scope of a universal one) this method may fail, and no bound can be derived.

6.2.5 A Detailed Example

To illustrate how complex data and time constants are dealt with, we solve a small example query in detail. We have a knowledge base KB containing two data items, namely

$$\begin{array}{l} holds_in(has(john, book_1), int(t_1, t_2)). \\ holds_in(has(mary, book_2), int(t_2, t_3)). \end{array}$$

We want to know if it is possible that, for arbitrary time points a,b and c,

$$holds(has(john, book_1), int(a, b)) \vee notholds(has(mary, book_2), int(a, c))$$

is true. The following query is used:

$$\begin{array}{ll} \leftarrow possible(KB, & [holds_in(has(john, book_1), int(a, b)) \vee \\ & notholds_in(has(mary, book_2), int(a, c))]). \end{array}$$

The program collects the data from *KB* and adds the query to it. All time constants are replaced by variables, and we obtain the following time table:

$$\begin{array}{lll@{\qquad}lll} t_1 & - & X_1 & a & - & A \\ t_2 & - & X_2 & b & - & B \\ t_3 & - & X_3 & c & - & C \end{array}$$

The goal we want to solve then becomes

$$\begin{array}{ll} \leftarrow & holds_in(has(john, book_1), int(X_1, X_2)), \\ & holds_in(has(mary, book_2), int(X_2, X_3)), \\ & [holds_in(has(john, book_1), int(A, B)) \vee \\ & \;\; notholds_in(has(mary, book_2), int(A, C))] \end{array}$$

but the complex data still need to be transformed. In this case only the disjunction is complex. New rules

$$\begin{array}{lll} q_0(A, B, C) & \leftarrow & holds_in(has(john, book_1), int(A, B)). \\ q_0(A, B, C) & \leftarrow & notholds_in(has(mary, book_2), int(A, C)). \end{array}$$

are added to the program, and we pass the following query on to SLDNFA:

$$\begin{array}{ll} \leftarrow & holds_in(has(john, book_1), int(X_1, X_2)), \\ & holds_in(has(mary, book_2), int(X_2, X_3)), \; q_0(A, B, C). \end{array}$$

If we ask for a solution with three events, SLDNFA replaces time variables with skolem constants, determines the order on these time constants, and abduces the necessary initiations and terminations to prove the goal (using the new rules for q_0 together with the general definitions of the knowledge base formulae as open logic program).

The solution contains three events e_1, e_2 and e_3 occurring at times new_1, new_2 and new_3, where $X_1 = A = new_1$, $X_2 = B = new_2$ and $X_3 = C = new_3$. The order on the time points is $new_1 < new_2$, $new_2 < new_3$. The initiations are

$$\begin{array}{l} initiates(e_1, has(john, book_1)). \\ initiates(e_2, has(mary, book_2)). \end{array}$$

and terminations are not necessary. The time table now looks like this:

$$\begin{array}{lll@{\qquad}lll} t_1 & - & new_1 & a & - & new_1 \\ t_2 & - & new_2 & b & - & new_2 \\ t_3 & - & new_3 & c & - & new_3 \end{array}$$

and we can read the following solution

$$\begin{aligned}
&happens(e_1, t_1),\ happens(e_2, t_2),\ happens(e_3, t_3)\\
&t_1 = a,\ t_2 = b,\ t_3 = c,\\
&t_1 < t_2,\ t_2 < t_3,\\
&initiates(e_1, has(john, book_1)),\\
&initiates(e_2, has(mary, book_2)).
\end{aligned}$$

which is of course only one of the many solutions found by the procedure.

6.2.6 Advanced Functionality

SLDNFA can be used as the basis for a lot of other tasks related to the knowledge base. We give two examples: a method for resolving inconsistency in the knowledge base, and a way of using the knowledge base for general problem solving, in particular planning (the latter being one of the major goals in [39]).

Resolving inconsistency

When inconsistency is detected in the knowledge base, SLDNFA can be used to propose modifications (i.e. deletions of data items) eliminating the inconsistency. Basically, this is done by allowing — in a consistency check — each data item to either be satisfied or to be marked as rejected. To this end we only need to add to the theory an open predicate $reject/1$, which takes as argument a term $t(P)$ representing a particular data item P. Each data item P can then be handled by either satisfying P or by abducing $reject(t(P))$. We illustrate the practical details in the following example.

Suppose we have three formulae P, Q and R as data. The program collects these data in a list $[P, Q, R]$, which is given — after Lloyd-Topor transformation — as a goal to SLDNFA. If SLDNFA returns with a solution, the data are consistent and there is no problem.

If no solution is found, an attempt can be made to reject some data such that the inconsistency disappears. In practice, this is achieved by allowing an alternative transformation of the definition of each data item in the preceding transformation step: instead of applying the Lloyd-Topor transformation, P is replaced by the atom $reject(t(P))$.

SLDNFA then solves $reject(t(P))$ instead of P. Since $reject/1$ is an open predicate, this is always possible. The result is that Δ contains an abduced fact $reject(t(P))$, plus a model for $[Q, R]$. The constraint P is dropped, which possibly resolves the inconsistency.

In further attempts every combination of formulae and rejected formulae is checked until a solution is found. Looking at the abduced $reject(t(P))$

facts, the user sees which constraints have been dropped to restore consistency. Of course more than one solution will in general exist, and the user can choose the "best" one, whatever that means to him. Alternatively, or in combination with a final decision by the user, restrictions on *reject* can be added to ensure that particular data are retained.

As an example of the basic procedure, a knowledge base containing

$$holds_in(has(john, book_1), int(t_1, t_2)).$$
$$holds_in(has(mary, book_1), int(t_1, t_2)).$$
$$holds_at(has(X, O), T),\ holds_at(has(Y, O), T) \rightarrow X = Y.$$

is inconsistent, and consistency can be restored by deleting any of the three constraints. Alternatively, rejection of "general" constraints (i.e. in this example the third one) would typically be prohibited. One of the proposed solutions would be

$$reject(t(holds(has(john, book_1), int(t_1, t_2))))).$$
$$happens(t_1). \quad happens(t_2). \quad t_1 << t_2.$$
$$initiates(t_1, has(mary, book_1)).$$

This method is of course inefficient even regardless the (in)efficiency of SLDNFA, since data are selected for rejection in a random way, without looking for the causes of the inconsistency. However there exist solutions to this problem, an issue to which we will return briefly in the discussion.

Planning

We now show how the knowledge base can be used for planning. Given our formalisation of the knowledge base in OLP Event Calculus, it is probably not surprising that planning is possible.

Of course we need to introduce the concept of an action, and define all possible actions with their preconditions and their effects. These effects are given in terms of a definition of *initiates* and *terminates*, as usual. For planning in Event Calculus, the predicate *act* is an open predicate. The representation of the knowledge base can be adapted accordingly. The difference with the original representation is that *initiates* and *terminates* are defined in terms of *act*, i.e. initiations and terminations are no longer arbitrary but must be caused by appropriate actions. This use of actions as a cause for every initiation and termination can also be extended to non-planning problems and in fact form an improvement of the knowledge base system as a whole: the actions define every possible way in which properties can change, ensuring that this happens in sensible ways. Abduced "models"

of our knowledge base would then not only contain information about which properties change value when, but also about *why* this happens.

Given an open *act* predicate and definitions of initiation and termination in terms of it, the setting has become identical to the usual Event Calculus setting for planning. Hence, the knowledge base can be used just like any Event Calculus theory. As an example, assume we want to build a very simple plan: John owns a certain book, and we want Mary to have it. The only possible action is giving the book to someone. We add the specification of this action's preconditions and effects to our basic definitions:

$$\begin{array}{lll} invalid & \leftarrow & act(E, give(Y, B, X)),\ happens(E, T), \\ & & \neg holds_at(has(Y, B), T). \\ initiates(E, has(X, B)) & \leftarrow & act(E, give(Y, B, X)). \\ terminates(E, has(Y, B)) & \leftarrow & act(E, give(Y, B, X)). \end{array}$$

Like in usual Event Calculus planning we also introduce a special event *start* which occurs before all other events to take care of the first initiations. After this special event, only actions can change the world.

$$\begin{array}{lll} happens(start, t_0). & & \\ invalid & \leftarrow & happens(E, T),\ T < t_0. \\ initiates(start, P) & \leftarrow & initially(P). \end{array}$$

and we assume *happens*, $<$, *initially* and *act* to be open.

The knowledge base KB contains the formulae

$$\begin{array}{l} holds_at(has(john, book_1), t_1). \\ holds_at(has(X, B), T) \wedge holds_at(has(Y, B), T) \rightarrow X = Y. \end{array}$$

and we try to solve the query

$$\leftarrow possible(KB, [t_1 < t_2,\ holds_at(has(mary, book_1), t_2)]).$$

This yields for example the model

$$\begin{array}{ll} happens(start, t_0). & \\ happens(e_1, t_1). & t_0 < t_1 \\ happens(e_3, t_3). & t_1 < t_3 \\ happens(e_2, t_2). & t_3 < t_2 \\ initially(has(john, book_1)). & \\ act(e_3, give(john, book_1, mary)). & \end{array}$$

which explicitly contains the plan: one action at t_3.

Note that due to the incomplete knowledge assumed in this knowledge base, one should take care that all relevant fluent values in some starting

state or possibly intermediate state are explicitly specified. If some part of for example the initial state is not specified, it is consistent to assume that the corresponding part of the desired end state was already true from the beginning (In the previous example, this could mean assuming that Mary had the book all the time by abducing the corresponding *initially* atom. However in that example the constraint that only one person can own a book at the same time eliminates this unintended solution.)

As a final remark, note that the knowledge base representation with only initiations and terminations is a special case of the proposal with actions, in which for each initiation or termination there is one action with precisely and only that effect.

6.2.7 Discussion

We have demonstrated how OLP Event Calculus can be used as a general framework for the representation and use of temporal knowledge bases containing incomplete data. Both time points and time intervals can be represented and reasoned with.

Abductive reasoning provides a straightforward way to generate models for a set of data. This allows us to check consistency and to answer queries. Complex data can be handled using a preceding transformation step, and deskolemisation allows us to represent time points that may be equal to each other.

We have indicated how more advanced types of functionality can be provided by SLDNFA, like resolving inconsistency in the knowledge base and planning using the knowledge base. This shows that the representation is sufficiently general for easy practical use.

In general, the proposed algorithms are not efficient, even though the constraint module for partial orders in SLDNFA eliminates one major source of inefficiency. However, recall that our main goal is providing a theoretical framework for representing incomplete temporal knowledge bases, and giving a number of simple algorithms to illustrate how such knowledge bases can be used and manipulated. These algorithms should be considered a starting point for research on more efficient implementations. For example, incremental model construction would be very important to avoid building a complete model of the knowledge base for each query.

For some applications, we can enhance the efficiency by adopting ideas found in the literature: our framework is sufficiently flexible to allow for plugging in existing techniques. As just one example, in [117] we find an algorithm for resolving inconsistency in a network of interval relations, based on the work in [2]. There, for each pair of intervals a list of possible

relations between these intervals is maintained. If ever no possible relations are left between any two intervals, the data are inconsistent. Weigel and Bleisinger have modified and extended this procedure to efficiently derive solutions for the inconsistency. They present two possible approaches: one using *elementary reasons* and one using *elementary solutions*.

The first approach attempts to find minimal sets of inconsistent constraints: starting from the initial inconsistent set, as many constraints as possible are removed while preserving inconsistency. This results in an *elementary reason*. One constraint from each elementary reason is then removed, eliminating one cause of the inconsistency. The process is repeated until consistency results.

A problem with this approach is that it often requires many steps, and too many constraints are eliminated. Therefore, another method is proposed as well. This method starts with an unrestricted set of relations, then tries to add as many constraints as possible without losing consistency. This results in an *elementary solution*. The constraints that could not be added are reported to the user.

The solutions show some similarity to our approach, but work only on interval relations instead of general data. This allows for more efficient algorithms, especially if an incremental consistency checker is used. By using a mapping that reduces general inconsistency to inconsistency in interval relations, which is possible if we use information like "intervals with incompatible properties must be disjoint", we could incorporate these techniques in SLDNFA. We do not pursue the issue further here.

An approach to the representation of temporal databases which is similar to ours can be found in [85]. A database is considered a collection of maximal intervals throughout which certain properties hold. For each property a list of such intervals is maintained. Incomplete information on the extent of intervals is represented by skolemising the end points of each interval, and constraints on these end points can be expressed. The framework shows some similarity to ours, though no explicit events are used and only maximal intervals are represented. The proposal can be mapped to ours, however, which would allow us to use its more efficient algorithms.

In conclusion, we want to indicate that the most important aspect of our framework is that it allows for the data in the knowledge base to be in the same language as the applications working with them. That this is achieved is clearly illustrated by the straightforward extension for planning. Thus we hope to show that the OLP Event Calculus is not only useful in several distinct temporal reasoning domains, but can also serve as a link bridging existing unnecessary gaps between them.

6.3 Protocol Specification

6.3.1 Introduction

In this section we use OLP Event Calculus for protocol specification. This is a research area far from the usual Event Calculus and artificial intelligence applications, and one which employs a range of very specialised formal specification languages. The most important of these are the process algebras CSP ([46]), CCS ([75]) and their descendants such as LOTOS ([106]), etc.. These languages are based on a mathematical abstraction of a process as an algebraic entity which can be constructed by combining basic processes using a class of pre-defined operators.

On the other hand, a distributed system of processes is just one example of a dynamic system, the area of application of the Event Calculus. Hence, using Event Calculus for protocol specification and comparing it with specialised existing approaches is a challenge which cannot be ignored and an important test of the practical applicability of the formalism.

In contrast to process algebras, where the concept of a process is hard-coded in the semantics, Event Calculus has no specialised representation of processes at all. We model a process as a dynamic entity with an identity (a name) and an internal state which is represented by a set of attributes or relations. A process has a restricted ability to sense actions executed by other entities (e.g. when receiving or synchronising on messages) and has the ability to execute actions (e.g. by sending messages). Both sensing and executing actions modify its internal state and a restricted part of the outer world (e.g. receiving a message deletes a message from a channel). In section 6.3.3, we adopt this dynamic view on processes in a specification of a communication protocol, the sliding window protocol with go-back-n from [105].

Our study is interesting from different perspectives. First, it illustrates the use of open logic programming and Event Calculus in a setting they were not initially intended for, thus illustrating their flexibility. Second, it shows that the classical temporal reasoning theories developed in artificial intelligence are growing out of their infancy and are ready for larger tasks. Third, it is interesting that the representation style of processes in our experiment is strikingly different from that in a process algebra specification. As argued in section 6.3.5, this is due to the differences between the algebraic view on processes and the view of a process as a dynamic entity with an identity and an internal state. The representation style we adopt will be argued to be more generally applicable, and therefore more suited for integration with other applications.

Though the view of a process as a dynamic entity fits in more naturally

with the ontological primitives of Event Calculus, the Event Calculus is sufficiently general to also represent the algebraic view on processes. This makes an integration of process algebra specifications in Event Calculus feasible. Section 6.3.4 presents a general technique to incorporate protocol specifications using process algebras in Event Calculus.

6.3.2 Preliminaries and Notation

Our specification basically uses the Event Calculus specified in Chapter 4. We introduce some additional concepts and notations specific to the protocol specification setting, to make the representation more concise.

In process protocols, we can generally assume that there are no simultaneous actions. Hence we impose

$$act(E, A_1) \wedge act(E, A_2) \rightarrow A_1 = A_2.$$

Many or all events in a process protocol have important preconditions. We use the *precondition* predicate to represent these: $precondition(A, T)$ means that a sufficient precondition for the action A to happen at time T is satisfied. We then have

$$happens(E, T) \wedge act(E, A) \rightarrow precondition(A, T)$$

The predicate *precondition* will be defined by domain specific rules.

Another issue is the representation of complex objects (in particular processes). One way of modeling a complex object in logic is by defining a number of predicates representing the relations in which the object may occur. An alternative is the object-oriented style which uses *attributes* to represent the knowledge on an object. Attributes are suitable to represent partial functions of complex objects and time. We have often found it useful and elegant to use a mixed representation for complex objects, using attributes for the partial functions, and using predicates for other types of relations.

Attributes are described by the fluent *attribute*. The atom

$$holds(attribute(P, PROP, VAL), T)$$

means that the object P has a value VAL for attribute $PROP$, at time T. This representation allows us to formalise the law of destructive assignment for all attributes: any initiation of a value for an attribute deletes the old value.

$$terminates(E, attribute(P, PROP, OLD_VAL)) \leftarrow \left| \begin{array}{l} initiates(E, attribute(P, PROP, VAL)), \\ happens(E, T), holds(attribute(P, PROP, OLD_VAL), T), \\ \neg OLD_VAL = VAL. \end{array} \right.$$

This axiom is assumed in the specification below.

In our specification, often a particular action occurring in a certain context has several effects. To limit lengthy repetitions, we will therefore adopt the notation

$$\begin{array}{c|c|c} A_1 & & B_1 \\ \vdots & \leftarrow & \vdots \\ A_n & & B_m \end{array}$$

to represent the set of clauses

$$A_1 \leftarrow B_1 \ldots B_m.$$
$$\vdots$$
$$A_n \leftarrow B_1 \ldots B_m.$$

6.3.3 Specification of a Sliding Window Protocol

In this section we give an example of a specification of a non-trivial protocol in Event Calculus : the so-called sliding window protocol with go-back-n. This is a well-known communication protocol, described in [105], which is situated in the OSI datalink-layer. The goal of the protocol is to provide a reliable connection to be used by the higher OSI layers, in particular the network layer, given an unreliable physical channel. A network layer process passes frames it needs to send to its underlying datalink layer process, which makes use of the sliding window protocol to pass them over the physical channel to the datalink process on the other side. This process in turn passes the frames to the desired network layer process on its side of the channel. The protocol is symmetrical (processes are both sender and receiver) and uses pipelining (multiple frames can be sent out before one is received on the other side) and piggybacked acknowledgements (acknowledgements are sent by the receiver encapsulated in its own messages going the other way). Flow control is based on a sending window which stores sent unacknowledged frames and a receiving window (in this case of length 1) of frames that can be accepted. If a particular frame is lost or corrupted, no other frames can be accepted until the lost one is resent: all subsequently sent frames then have to be resent as well, hence the name "go-back-n".

We give a more detailed description of the protocol. Essentially, a process in the datalink layer waits for and reacts to various incoming input events. Three types of input events exist:

- a send event: a network layer process passes a frame to the datalink process. The latter process puts the frame in a slot in its sending win-

dow, initiates the timer associated with that slot, and sends a packet including the frame and some additional information out on the physical channel. This additional information consists of the number of the sent frame's slot and an acknowledgement indicating the number of the last frame correctly received from the other side.

- a **receive** event: a packet arrives over the physical channel. If it does not contain the expected frame (i.e. its slot number is not the successor of the last received packet's slot number) or if the packet is corrupted, the packet is rejected without further processing. Otherwise, the packet is accepted and the frame it carries is passed to the network layer. Also, the acknowledgement number carried by the packet indicates that all frames up to that number have been received by the peer process, hence all slots in the sending window up to the acknowledgement number can be cleared and all associated timers turned off.
- a **timer** event: the timer of (the oldest) frame in the sending window is ringing. The process goes into a mode in which it retransmits its buffer, i.e. the contents of all slots in its sending window, over the channel. After this is done, the process waits again for an input event.

In the following sections, this informal specification will be used as the basis for a formal specification in Event Calculus. In this specification the predicates *happens*, *act* and $<$ are open predicates, since the protocol specifies only *potential* behaviour and no actual scenario: the protocol determines only which events *can* occur under which circumstances.

General concepts

We first describe and specify the environment in which the protocol operates: the main types of objects in the domain, with their attributes and relations. The main types of objects are the datalink layer processes, the channel, the frames and of course the events which take place. There are two peer processes (both are sender/receiver), which are connected by a channel. The two processes are denoted p_1 and p_2. p_1 and p_2 are each other's receiver process. The following clauses represent this:

$process(p_1)$.
$process(p_2)$.
$receiver(p_1, p_2)$.
$receiver(p_2, p_1)$.

The attributes of a process are listed below:

- *mode*: this attribute can take on four different values:
  - *input*: the process is ready and waiting for input
  - *sending*(F, NR): the process is sending a frame F stored in the slot NR
  - *receiving*(F): the process is passing a received frame F to its network layer
  - *retransmitting*(NR): after a time-out, the process is retransmitting its buffer starting from slot number NR.
- *xpf*: the *frame to be received* (*expected frame*) attribute: this attribute points to the number of the next frame to be received, i.e. the frame with number one higher than the last successfully received frame.
- $w(i)$ (for any $0 \leq i < n - 1$ with n the size of the window): the value of the attribute w_i of process P is the frame which is stored at slot i of the window. The window is a circular buffer of length n. This is represented by the predicate *cnext* which is defined as follows:

  $cnext(0, 1)$.
  $cnext(1, 2)$.
  ...
  $cnext(n - 1, 0)$.

- *xpa*: the *oldest frame to be acknowledged* (*expected acknowledgement*) attribute: this attribute takes values $0 \leq i < n$ and points to the oldest sent but unacknowledged frame in the sending window.
- *fts*: the *frame to send* attribute: this attribute takes values $0 \leq i < n$ and points to the first free slot in the sending window.

A fluent on processes is *networklayer_enabled*(P), meaning that the network layer of process P is allowed to pass new frames to P.

We distinguish between *frames* and *packets*: a frame is a data item given by a process in the network layer to the datalink process for transmission. A packet is a physical unit of data which is sent over the channel. Two different frames or two different packets may carry precisely the same information, and two different packets may also contain the same frame. We need to distinguish between packets and frames carrying the same information but created at different instants in time in order to elegantly express some of the physical properties of the system. Take, for example, the property that if one packet is received earlier than another packet, then the first packet was also sent earlier than the second. In this sentence,

packets should clearly not be interpreted as the data they contain, but as a particular occurrence of these data.

How can we characterise frames and packets ? A frame is introduced in the system by the event E in which a network layer process passes the frame to P, so we can denote the frame by the term $frame(E, P)$. A packet contains as information a frame, a slot number and an acknowledgement, and can hence be denoted by $packet(E, FRAME, NR, ACK)$. Here E is the send event by which the packet is created and sent over the channel. $FRAME, NR, ACK$ are the data carried in the packet: $FRAME$ is the carried frame, NR the slot number in which this frame is stored and ACK the number of the last successfully received frame.

Packets appear in two different relations in the specification:

- $corrupt(PACKET)$: the packet $PACKET$ has been corrupted by errors of the channel;
- $on_channel(PACKET, P)$: the packet $PACKET$ is on its way on the physical channel toward receiver process P.

Next, we specify the possible events in the specification. There are 7 event types. The first four types model the sending and receiving actions of datalink and network layer processes. One event type models a timer run-out. The last two types model two different errors which can occur on the channel:

- $net_send(F, P)$: the network layer passes the frame F to its datalink layer process P.
- $net_receive(P, F)$: the datalink process P passes a received frame F to the network layer;
- $send(P, PACKET)$: P sends packet $PACKET$ out on the channel.
- $receive(P, PACKET)$: P receives packet $PACKET$ from the channel.
- $timer_rings(P, NR)$: the timer of slot $w(NR)$ of process P rings;
- $disturbance_channel(PACKET)$: a disturbance on the channel corrupts packet $PACKET$;
- $failure_channel(PACKET)$: a failure of the channel causes the loss of packet $PACKET$.

Initial state of the system

The following clauses define the initial state, in which both processes are in input mode, have an empty sending window, and have enabled network layers:

$initially(attribute(p_1, mode, input)).$
$initially(attribute(p_2, mode, input)).$
$initially(attribute(p_1, fts, 0)).$
$initially(attribute(p_1, xpa, 0)).$
$initially(attribute(p_1, xpf, 0)).$
$initially(attribute(p_2, fts, 0)).$
$initially(attribute(p_2, xpa, 0)).$
$initially(attribute(p_2, xpf, 0)).$
$initially(networklayer_enabled(p_1)).$
$initially(networklayer_enabled(p_2)).$

The channel

We start by describing the channel. A fundamental assumption about the physical channel is that it preserves the order of packets. The following definition of *invalid* formulates this:

$$invalid \leftarrow \left| \begin{array}{l} happens(E_1, T_1), happens(E_2, T_2), T_1 < T_2, \\ act(E_1, send(PROCESS_1, PACKET_1)), \\ act(E_2, send(PROCESS_1, PACKET_2)), \\ receiver(PROCESS_1, PROCESS_2), \\ act(E_3, receive(PROCESS_2, PACKET_1)), \\ act(E_4, receive(PROCESS_2, PACKET_2)), \\ happens(E_3, T_3), happens(E_4, T_4), T_4 < T_3. \end{array} \right.$$

Two types of events are local to the channel. These are the events which simulate errors of the channel. Their preconditions and effects are described below:

- A disturbance on the channel corrupts a packet. This can of course only happen to packets on the channel:

  $precondition(disturbance_channel(PACKET), T)$
  $\leftarrow process(PROC), holds(on_channel(PACKET, PROC), T).$

  Its effect is that the packet becomes corrupted:

  $$initiates(E, corrupt(PACKET)) \leftarrow act(E, disturbance_channel(PACKET)).$$

- A failure of the channel which causes the loss of a packet. Precondition is that the packet is on the channel:

  $$precondition(failure_channel(PACKET), T) \leftarrow process(PROC), holds(on_channel(PACKET, PROC), T).$$

  Its effect is that the packet is removed from the channel:

  $$terminates(E, on_channel(PACKET, PROCESS)) \leftarrow act(E, failure_channel(PACKET)).$$

Sending behaviour of a process

For the description of a process, we follow the structure of the informal specification of the protocol in section 6.3.3. Assume that the process receives a frame from the network layer. Preconditions for this event to happen are that the peer process is in input mode and that the network layer is enabled. The passed frame is given a name determined by the *net_send* event and the process.

$$precondition(net_send(frame(E, PROCESS), PROCESS), T) \leftarrow \left| \begin{array}{l} process(PROCESS), happens(E, T), \\ holds(attribute(PROCESS, mode, input), T), \\ holds(networklayer_enabled(PROCESS), T). \end{array} \right.$$

The effect of a $net_send(FRAME, PROCESS)$ event is that the process enters sending mode, that the frame is stored in the first free slot of the sending window, as indicated by the *fts* (frame to send) attribute, that *fts* is incremented, and that if the buffer is full, the network layer is disabled.

$$\left. \begin{array}{r} initiates(E, attribute(PROCESS, mode, sending(FRAME), FTS))) \\ initiates(E, attribute(PROCESS, w(FTS), FRAME)) \\ initiates(E, attribute(PROCESS, fts, NEXTFTS)) \end{array} \right| \leftarrow \left| \begin{array}{l} happens(E, T), act\ (E, net_send(FRAME, PROCESS)), \\ holds(attribute(PROCESS, fts, FTS), T), \\ cnext(FTS, NEXTFTS). \end{array} \right.$$

As mentioned, the network layer is disabled when the process is out of buffer space. In fact, to ensure correctness of the protocol (i.e. to avoid data loss) it must be disabled even sooner, when there is only one free buffer slot left (see [105] for more details). There is only one buffer slot left when

fts points to the predecessor of *xpa*. This is expressed by:

$$terminates(E, networklayer_enabled(PROCESS)) \leftarrow \left| \begin{array}{l} happens(E,T), \\ initiates(E, attribute(PROCESS, fts, FTS)), \\ holds(attribute(PROCESS, xpa, XPA), T), \\ cnext(FTS, XPA). \end{array} \right.$$

An implicit effect of the *net_send* event is that the timer of the slot of the frame is set. Here we simply assume that the timer of a slot is set as long as the slot is in the active part of the sending window; or, from the moment a frame is stored in the slot until an acknowledgement for the slot is received.

The only event that a process can execute when it is in sending mode is sending the frame, together with its slot number and an acknowledgement. The precondition of the send event is that the process is in sending mode, that the packet carries the correct data (frame, slot number and acknowledgement), and that the packet is the one uniquely corresponding to this event:

$$precondition(send(PROCESS, packet(E, FRAME, NR, ACK), T) \leftarrow \left| \begin{array}{l} process(PROCESS),\ happens(E,T), \\ holds(attribute(PROCESS, mode, sending(FRAME, NR)), T), \\ holds(attribute(PROCESS, xpf, XPF), T), \\ cnext(ACK, XPF). \end{array} \right.$$

An obvious effect of sending is that the packet gets on the channel.

$$initiates(E, on_channel(PACKET, RECEIVER)) \leftarrow \left| \begin{array}{l} act(E, send(PROCESS, PACKET)), \\ receiver(PROCESS, RECEIVER). \end{array} \right.$$

Another effect of sending in this mode is that the process returns to input mode.

$$initiates(E, attribute(PROCESS, mode, input)) \leftarrow \left| \begin{array}{l} happens(E,T),\ act(E, send(PROCESS, PACKET)), \\ holds(attribute(PROCESS, mode, sending(FRAME, NR)), T). \end{array} \right.$$

Receiving behaviour of a process

When a process is in input mode, it may receive packets from the channel. A sufficient precondition for this event to occur is that the process is in input mode and that the packet is on the channel:

$$\begin{array}{l}
precondition(receive(PROCESS, PACKET), T) \\
\leftarrow \left| \begin{array}{l} process(PROCESS), \\ holds(attribute(PROCESS, mode, input), T), \\ holds(on_channel(PACKET, PROCESS), T). \end{array} \right.
\end{array}$$

The effect of this event depends on several factors and is split up in different rules. One effect of receiving is always that the frame is removed from the channel:

$$\begin{array}{l}
terminates(E, on_channel(PACKET, PROCESS)) \\
\leftarrow \left| \; act(E, receive(PROCESS, PACKET)). \right.
\end{array}$$

To be accepted, a received packet should not be corrupted and should carry the expected frame number given by attribute xpf. If these conditions are satisfied, the xpf attribute is circularly increased and the process enters the receiving mode during which the frame is passed to the network layer.

$$\begin{array}{l}
initiates(E, attribute(PROCESS, xpf, NEXTXPF)) \\
initiates(E, attribute(PROCESS, mode, receiving(FRAME))) \\
\leftarrow \left| \begin{array}{l} happens(E, T), \; act(E, receive(PROCESS, PACKET)), \\ \neg holds(corrupt(PACKET), T) \\ PACKET = packet(E', FRAME, NR, ACK), \\ holds(attribute(PROCESS, xpf, XPF), T), NR = XPF, \\ cnext(XPF, NEXTXPF). \end{array} \right.
\end{array}$$

A second part of the receiving behaviour is the handling of the acknowledgement. When an uncorrupted packet is received and it carries an acknowledgement ACK which corresponds to some currently used slot of the sending window, then all slots between (and including) the expected acknowledgement (xpa) and the slot ACK are released by setting the xpa attribute to the successor of ACK. Due to the nature of the protocol, this is safe even when the received acknowledgement ACK is not the expected acknowledgement.

In addition, if the network layer was disabled, it will now be enabled

again. This is formalised in the following clauses.

$$\begin{array}{l} initiates(E, attribute(PROCESS, xpa, NEXTXPA)) \mid \\ \quad initiates(E, networklayer_enabled(PROCESS)) \mid \\ \leftarrow \left| \begin{array}{l} happens(E,T), act(E, receive\ (PROCESS, PACKET)), \\ \neg holds(corrupt(PACKET), T), \\ PACKET = packet(E', FRAME, NR, ACK), \\ holds(attribute(PROCESS, xpa, XPA), T), \\ holds(attribute(PROCESS, fts, FTS), T), \\ c_between(XPA, ACK, FTS), \\ cnext(ACK, NEXTXPA). \end{array} \right. \end{array}$$

The predicate $c_between(X, I, Y)$ denotes that I is circularly between X and Y (including X but not Y):

$$\begin{array}{ll} c_between(X, X, Y) \leftarrow & X \neq Y. \\ c_between(X, I, Y) \leftarrow & X \neq Y, cnext(X, X1), c_between(X1, I, Y). \end{array}$$

When a process is in receiving mode, it passes the received frame to the network layer. The precondition of the action *net_receive* is that the process is in receiving mode.

$$\begin{array}{l} precondition(net_receive(PROCESS, FRAME), T) \\ \leftarrow \left| \begin{array}{l} process(PROCESS), \\ holds(attribute(PROCESS, mode, receiving(FRAME)), T). \end{array} \right. \end{array}$$

The effect of *net_receive* is that the process returns to input mode:

$$\begin{array}{l} initiates(E, attribute(PROCESS, mode, input)) \mid \\ \leftarrow \mid act(E, net_receive(PROCESS, FRAME)). \end{array}$$

Handling of a ringing timer

Finally we show how ringing timers are handled. We know that whenever a timer rings, the buffer is non-empty and, since all timers have an equal timing interval, the timer corresponds to the oldest frame in the sending window. This yields the following precondition for *timer_rings*:

$$\begin{array}{l} precondition(timer_rings(PROCESS, XPA), T) \\ \leftarrow \left| \begin{array}{l} process(PROCESS), \\ holds(attribute(PROCESS, mode, input), T), \\ holds(attribute(PROCESS, xpa, XPA), T), \\ holds(attribute(PROCESS, fts, FTS), T), \\ \neg XPA = FTS. \end{array} \right. \end{array}$$

When this timer rings, all unacknowledged frames have to be retransmitted. The process enters retransmitting mode with as parameter the expected acknowledgement.

$$initiates(E, attribute(PROCESS, mode, retransmitting(XPA))) \\ \leftarrow \mid act(E, timer_rings\ (PROCESS, XPA)).$$

While a process is in retransmitting mode, it sends out its buffer. This situation yields a second sufficient precondition for a send event: the process is in retransmitting mode and the sent frame is the one pointed to by the parameter of the mode:

$$\begin{array}{l} precondition(send(PROCESS, packet(E, FRAME, NR, ACK)), T) \\ \leftarrow \left| \begin{array}{l} process(PROCESS), \\ holds(attribute(PROCESS, mode, retransmitting(NR)), T), \\ holds(attribute(PROCESS, w(NR), FRAME), T) \\ holds(attribute(PROCESS, xpf, XPF), T), \\ cnext(ACK, XPF). \end{array} \right. \end{array}$$

The effects of sending in retransmitting mode are slightly different from those in sending mode. The additional effect is that either the number of the slot to be retransmitted is incremented or, when the last frame of the buffer is retransmitted, that the process returns to input mode:

$$\begin{array}{l} initiates(E, attribute(PROCESS, mode, retransmitting(NEXTNR))) \\ \leftarrow \left| \begin{array}{l} happens(E, T), act(E, send\ (PROCESS, PACKET), \\ holds(attribute(PROCESS, mode, retransmitting(NR)), T), \\ cnext(NR, NEXTNR), \\ holds(attribute(PROCESS, fts, FTS), T), \\ \neg FTS = NEXTNR. \end{array} \right. \end{array}$$

$$\begin{array}{l} initiates(E, attribute(PROCESS, mode, input)) \\ \leftarrow \left| \begin{array}{l} happens(E, T), act(E, send(PROCESS, PACKET)), \\ holds(attribute(PROCESS, mode, retransmitting(NR)), T), \\ holds(attribute(PROCESS, fts, FTS), T), \\ cnext(NR, FTS). \end{array} \right. \end{array}$$

This concludes the specification of the sliding window protocol with go-back-n. It is our task to prove that this specification is correct and meets the requirements of the protocol, i.e. in particular that the network layer on one side will receive all frames sent out on the other side in the correct order and only once (or will receive nothing at all from a certain moment on if the channel breaks down entirely). We prove this property of the protocol (by hand) in Appendix C.

6.3.4 Translating Process Algebra Specifications to Event Calculus

In this section, we show how any specification formalism of which the semantics can be defined in terms of a labeled transition system by means of a logic program, can easily be translated to OLP Event Calculus. In particular, it follows that process algebras like LOTOS, CCS and CSP can be translated to OLP Event Calculus.

The purpose of such a translation is twofold. First, it shows that the Event Calculus has at least the expressive power of process algebras. Second, the existence of such a straightforward translation makes the process of combining specifications in different languages easier: if one is building an Event Calculus specification and wants to reuse existing specifications written in a process algebra, these existing specifications can be incorporated in the Event Calculus specification by translating them as explained in this section. Of course, the translated specifications will not be very well-structured from an Event Calculus point of view, but all existing tools for reasoning on the Event Calculus can be used on them.

In process algebras, a process is defined as the set of all possible sequences of actions it can consist of. This set is written as a ***possible behaviour expression***. Labeled transition systems are widely used to give a semantics to such possible behaviour expressions. A ***labeled transition system*** consists of two sets (a set S of states and a set L of transition labels), and a relation on $S \times L \times S$, called the transition relation. Usually, L is interpreted as a set of actions, and (s_1, l, s_2) is in the transition relation if the specified system can go from state s_1 to state s_2 by performing action l. For example, the semantics of LOTOS ([106]) is defined by means of a labeled transition system, where S is the set of possible behaviour expressions and L is the set of possible actions.

A labeled transition system can be specified in a logic program by defining a predicate *transition*/3. For example, we can define the semantics of basic LOTOS behaviour expressions, built using the process constant **stop**, the prefix operator ";" which concatenates an action and a behaviour expression, the choice operator "[]" which denotes a choice between two behaviour expressions, and the parallel composition operator between behaviour expressions "$|G|$" (where G is a list of actions which must be performed synchronously by the component expressions). This definition is given by the following logic program:

$$\begin{array}{l} transition(A; S, A, S). \\ transition(S_1 [] S_2, A, S_1') \leftarrow transition(S_1, A, S_1'). \\ transition(S_1 [] S_2, A, S_2') \leftarrow transition(S_2, A, S_2'). \end{array}$$

$$transition(S_1|G|S_2, A, S_1'|G|S_2) \leftarrow \left| \begin{array}{l} transition(S_1, A, S_1'), \\ A \notin G. \end{array} \right.$$

$$transition(S_1|G|S_2, A, S_1|G|S_2') \leftarrow \left| \begin{array}{l} transition(S_2, A, S_2'), \\ A \notin G. \end{array} \right.$$

$$transition(S_1|G|S_2, A, S_1'|G|S_2') \leftarrow \left| \begin{array}{l} transition(S_1, A, S_1'), \\ transition(S_2, A, S_2'), \\ A \in G. \end{array} \right.$$

where the $\in$ predicate needs to be defined appropriately depending on the representation of G. It is straightforward to define the transition predicate for other process algebra connectives in the same way.

Once we have defined the transition predicate for a specific process algebra, the translation to Event Calculus is easy. Suppose we have an arbitrary labeled transition system defined by a predicate $transition(S_1, A, S_2)$ and with initial state s_1, then the following clauses give a correct Event Calculus description of the system:

$$initiates(E, STATE) \leftarrow \left| \begin{array}{l} happens(E, T), holds(OLDSTATE, T), \\ transition(OLDSTATE, A, STATE), \\ act(E, A). \end{array} \right.$$

$$terminates(E, STATE) \leftarrow \left| \begin{array}{l} happens(E, T), holds(STATE, T), \\ transition(STATE, A, NEWSTATE), \\ act(E, A), \\ \neg STATE = NEWSTATE. \end{array} \right.$$

$$initially(s_1).$$

These rules are a direct formalisation of the intended meaning of a labeled transition system: each action terminates the current state and initiates the new state determined by the *transition* relation.

6.3.5 Discussion

Process algebras provide support for the representation of processes and synchronisation in the sense that these notions are hard-coded in the semantics of these languages. In contrast, OLP Event Calculus is a universal logic for representing changing worlds; it does not provide hard-coded concepts like processes or synchronisation. This observation might lead one to expect a high verbosity in our specification, due to the lack of support of

central concepts. However, the verbosity in our specification is surprisingly low. Our specification contains about 30 main domain dependent clauses (not counting e.g. the initially clauses and the definition of cnext) with an average of 3-4 literals in the body. The specification in section 6.3.3 and the specification of the same protocol in a process algebra in [53], have about the same length.

Comparing our specification with a specification in a process algebra, we find an important difference in the conceptualisation of the *process* concept. In a process algebra a process is a static, algebraic entity built up by combining simpler processes using pre-defined operators. In contrast, the Event Calculus specification models a process as a dynamic entity with an identity and an internal state. A process has attributes representing the internal state, it has a restricted ability to sense actions executed by other entities and has the ability to execute actions which modify its internal state and a restricted part of the outer world.

The differences in view lead to extremely different specification styles. The Event Calculus specification models the state of the world at each moment: packets that are on the channel, contents of the slots in the sending window, current mode of the processes, expected frame and acknowledgement numbers, and so on. It also describes the evolution of the world as a result of events, and the preconditions of each event type. From this information, possible sequences of events (*traces*) can be derived.

Process algebra specifications do not model the evolving state of the world, but only and immediately the set of possible traces that is derived from it. On the one hand, this is an advantage if the traces are the only thing one is interested in: a lot of unnecessary information is abstracted away. On the other hand the loss of information severely limits the applicability of process algebras.

Another consideration is that in our view Event Calculus style specifications, due to the fact that they model the real-world parameters from which traces can be derived instead of the traces themselves, tend to be both easier to produce and easier to modify — although admittedly process algebra experts disagree with this — : preconditions and effects can be described for each event type independently, whereas calculating the possible traces requires taking into account all possible interactions. Moreover, changes in the effects of one event can have a considerable and complicated influence on the set of possible traces, where it only leads to the modification of one effect clause in an Event Calculus style specification.

So far, we have focused on the role of open logic programming for specification. It is now time for a few words on reasoning on these specifications.

For Lotos specifications, software tools have been developed for differ-

ent computational tasks, including testing and simulation, verification and compilation of the specification in executable programs (for an overview see [8]). Tools performing similar tasks can be developed or exist already for the Event Calculus. For example in [78], a general purpose approach to simulation in Event Calculus is proposed. Other tasks can be performed on the basis of SLDNFA, as we sketch below.

Different forms of protocol verification are theoretically possible in OLP Event Calculus. For example, one requirement of the protocol specified in this section is that it should provide a perfect channel to the network layer, i.e. that frames sent by the network layer on one side arrive all exactly once and in the right order on the other side. Proving this is essentially a deductive problem.

Another typical verification problem in the context of distributed systems is whether a protocol is deadlock-free. A (simplified) condition which expresses that a deadlock arises at time T is that no event can happen at time T, i.e. that for no action its preconditions are satisfied. The following formula Ψ_d expresses this:

$$\exists T : \forall A : \neg precondition(A, T)$$

A deadlock-free protocol should entail $\neg\Psi_d$. This is also a deductive problem. In case the specification does not entail $\neg\Psi_d$, a third type of problem arises: namely to explain why Ψ_d can be true, i.e. how a deadlock may arise. This is a form of diagnosis, essentially an abductive problem. SLDNFA can be used to find a scenario (described in terms of the open predicates *happens*, $<$ and *act*) in which Ψ_d is true. One may observe that this problem is formally equivalent to a planning problem in Event Calculus.

Proofs of the correctness and deadlock-freeness of the protocol are included in Appendix C. These proofs have been generated by hand. Their complexity in combination with the inefficiency of the SLDNFA procedure show that more research in and specialised support for automated protocol verification are required, for example support for generating proposals of invariant relations. These are issues for further work which are not handled in this thesis.

The general applicability of Event Calculus in a wide range of tasks in dynamic systems allows for the reuse of protocol specifications for other tasks. For example, in this section we have specified a communication protocol. This protocol will typically be used by processes exchanging information in a network. An application in the same domain is network management and diagnosis. Obviously, both network diagnosis and protocol specification require knowledge of low-level parameters of the network, the communication channels, the states of processes and the occurring events.

For example, network diagnosis may need information on the frequency of packet losses on a particular channel, the average number of frames in unacknowledged slots, or the average number of retransmissions: information which can be obtained from the given specification. If different special purpose languages are used to deal with protocols on the one hand (e.g. process algebras), and the network diagnosis on the other hand (e.g. logic programming), then integration of and cooperation between these components becomes extremely difficult. It is a considerable advantage of OLP that it provides one general description language for the entire system, which can be used as the underlying specification language for most (or all) of its applications.

Chapter 7

A High-Level Language for Representing Dynamic Domains

7.1 Introduction

In the previous chapters we have illustrated the expressive power of OLP Event Calculus as a knowledge representation language useful in various problem domains where time is an important factor. In this chapter we return to the classical AI temporal reasoning setting to show how a language based on OLP Event Calculus can tackle the various aspects of the frame problem.

In itself, the OLP Event Calculus is not the best choice for this task: its expressiveness and flexibility, which are a considerable advantage when dealing with various domains, are also a source of risk if not handled with care. The Event Calculus deals only correctly with the frame problem if its theories are constructed following a particular methodology (like the one we have used throughout this thesis). For example, if a user would add clauses to the definition of *holds*, or if the definitions of $initiates(e, f)$ and $terminates(e, f)$ would contain statements about later time points than e or would not be given as a definition but as FOL axioms, etc., theories would be obtained which do not at all represent the intended knowledge. But since the Event Calculus is presented as an open logic program, there are no restrictions enforcing that the full expressive power of open logic programming is not (ab)used.

To cure this problem, we intend to develop a formalism which only allows one to represent theories in a form that yields the intended conclusions. At the same time, we of course want the formalism to retain as much of the expressive power of the OLP Event Calculus as possible, for example its observed ability to deal with incomplete scenario knowledge, with nondeterminism, with simultaneous actions and with indirect effects of actions. In other words, we want to isolate the language constructs responsible for the Event Calculus's expressive power and combine them in a high-level language for temporal knowledge representation. As the most difficult open problem in this research area is the ramification problem ([72],[72],[44]; see also Chapter 4), tackling this problem will be the issue receiving most attention.

Our approach follows the recently emerged trend of using high-level action languages for studying the general principles underlying time and change in certain well-defined settings. The first of these languages to emerge, the $\mathcal{A}$ language of [37], models inertia and direct effects of actions in a branching time topology, with possible uncertainty on the initial state of the world. Extensions of $\mathcal{A}$ tackle gradually more complex issues: for example $\mathcal{AR}_0$ ([55]) deals with indirect effects of actions (ramifications) and simple forms of nondeterminism. Another formalism, the $\mathcal{E}$ language ([50]), uses an event-based ontology modelled after a variant of the Event Calculus. $\mathcal{E}$ allows for modeling uncertainty on the initial state of the world and includes an initial idea on dealing with some ramifications, which is currently being developed further. The authors have also devised extensions of $\mathcal{E}$ to represent scenarios with incomplete knowledge on action ordering or action occurrences.

The language we design in this chapter will be named $\mathcal{ER}$. Though it is based, like $\mathcal{E}$, on a variant of the Event Calculus, it has little formal correspondence with $\mathcal{E}$. The main goal of the language is to correctly represent a very general set of indirect effects (ramifications), both of consecutive and simultaneous actions. Moreover the language is intended to deal in a flexible way with complete and/or incomplete knowledge on action occurrences, action ordering or the initial state. The language is also further extended to deal with nondeterminism and delayed effects of actions.

We argue that to represent all ramifications and qualifications of actions, it is necessary to include in the language *state constraints* as well as effect propagation rules (*derived effect rules*[1]) and explicit *action preconditions*: these three types of formulae are at least in part independent, i.e. there are derived effect rules that do not correspond to any state constraint and vice

[1] similar to "causal laws" or "causal rules" in the literature; we will use these terms as synonyms except where otherwise indicated

versa, and there are action preconditions not related to a state constraint. We motivate this and indicate differences between our approach and those in the recent literature.

We also argue that in many applications a clear and natural representation of indirect effects of actions in general and of the effects of simultaneous actions in particular can be obtained by using complex derived effect rules, i.e. effect rules stating that a change is triggered by the change in truth value of a *complex fluent formula*. $\mathcal{ER}$ includes such complex derived effect rules. A complete treatment of simple and complex derived effect rules requires relying on a strong semantics like that of open logic programs. We base the semantics of $\mathcal{ER}$ on the principle of inductive definitions, on which also the justification semantics is based. This principle yields at the same time the required expressiveness to deal with the frame and ramification problems even in the presence of negative and possibly cyclic dependencies between effects, while it has the advantage that the intuitions underlying the formal semantics (i.e. inductive definitions) are generally well-understood. Moreover we show that for restricted classes of definitions (like loop-free definitions or definitions without negations) for which simpler semantics have been proven adequate, the inductive definition semantics coincides with these semantics.

We assume in the first sections of this chapter that actions are deterministic, have no duration, and can be simultaneous, and that all changes are discrete. In this setting we intend to deal correctly with all immediate ramifications, i.e. all ramifications occurring at the time of the action(s) they are ramifications of. Moreover we deal correctly with action preconditions (qualifications) that are entailed by the theory. We do not handle default qualifications, as dealing with defaults is an entirely different problem than the inertia and ramification problems and not typical for temporal domains.[2] Once the basic language is established, we further extend $\mathcal{ER}$ to deal with delayed ramifications and with nondeterministic actions and ramifications.

A number of the issues tackled by $\mathcal{ER}$ have already been addressed in one or more other approaches to representing actions. In $\mathcal{ER}$ we want to tackle all of these issues in one coherent framework and at the same time address some unsolved problems. Meanwhile we want to keep the formalism and its intuitive (if not the formal) semantics simple. As we are dealing with a kind of "common sense" reasoning and representation, in our view the best way to meet these goals is to design the language with the general principle in mind to stay as close as possible to the intuitions about "real" time, actions, change and causality. This principle motivates, among others, the

[2] Ideas to deal with the default qualification problem can be found in [108].

decisions of representing a large part of the theory as simple first order logic (there is no reason for any more complicated choice), representing effects of actions as an inductive, constructive definition (which is how we intuitively interpret them) and choosing a time topology of actual events occurring on a single time line (as *real* time is most naturally seen as just one infinite line)[3].

Once we have developed $\mathcal{ER}$, we close the circle started above by providing a mapping of $\mathcal{ER}$ theories to a variant of full OLP Event Calculus, and by proving the correctness of this mapping.

We extensively compare the $\mathcal{ER}$-approach to the ramification problem with the one in [109], which shows most similarities to it. Finally, we study the idea introduced in [109] of using influence information in a tool for (semi-)automatically deriving causal laws from state constraints. We propose an alternative approach to this problem in the setting of $\mathcal{ER}$ and we illustrate the differences with Thielscher's proposal.

In the next section, we present the syntax of $\mathcal{ER}$ and motivate the design of the language in much detail. Section 7.3 discusses how $\mathcal{ER}$ tackles the ramification problem and defines the semantics of $\mathcal{ER}$. Section 7.4 contains a couple of detailed examples and sheds some light on particular contributions. A mapping to OLP Event Calculus is presented and proven correct in section 7.5. In section 7.6 we extend the language to deal with non-deterministic actions and ramifications. Subsequently a comparison with Thielscher's approach is provided in section 7.7, and a method for using influence information in section 7.8. Delayed effects of actions are introduced in section 7.9. In the two final sections we discuss more related work and conclude.

7.2 The Syntax of $\mathcal{ER}$

Basically, an $\mathcal{ER}$-theory consists of a set of effect rules determining direct and indirect effects of actions (a theory of causation), combined with a general first order theory describing the truth of fluents at certain times, the occurrence and order of actions, and general state constraints and action preconditions. Formally we define the following syntax :

Definition 7.2.1 **($\mathcal{ER}$-signature)**
An $\mathcal{ER}$-signature Σ is a tuple $<$**Sorts, Functors, Vars**$>$ *with*

- **Sorts** $= \{\mathcal{T}, \mathcal{A}, \mathcal{F}, \mathcal{P}\}$, *representing the sorts time, action, fluent and atom.*

[3] However, for a discussion on the choice between a linear and a branching time topology we refer back to Chapter 5, where this issue has been discussed in much detail.

- **Functors** *consists of*
  - *a set* $\mathbf{T}$ *of constants of sort* $\mathcal{T}$, *denoted* $t, t_1, \ldots,$ *which includes all real numbers;*
  - *a set* $\mathbf{A}$ *of constants of sort* $\mathcal{A}$, *denoted* $a, a_1, \ldots;$
  - *a set* $\mathbf{F}$ *of constants of sort* $\mathcal{F}$, *denoted* $f, f_1, \ldots;$
  - *four typed predicate symbols*[4]*:*
    Happens $: \mathcal{A} \times \mathcal{T} \rightarrow \mathcal{P};$
    $\leq: \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{P};$
    Initially $: \mathcal{F} \rightarrow \mathcal{P};$
    Holds $: \mathcal{F} \times \mathcal{T} \rightarrow \mathcal{P}.$
- $\mathbf{Vars} = \mathbf{Vars}_{\mathcal{A}} \cup \mathbf{Vars}_{\mathcal{T}}$, *disjoint infinite sets of variables of sort* $\mathcal{A}$ *resp.* $\mathcal{T}$, *denoted as* $A, A_1, \ldots$ *resp.* $T, T_1, \ldots$.

Definition 7.2.2 (terms, fluent formulae, general atoms) *Terms are constants or variables. Terms of sort* $\mathcal{T}$ *will be denoted by* τ, *action terms by* α. *A fluent literal* l *is either a fluent constant* f *or its negation* $\neg f$. *We define* $\widehat{\mathbf{F}}$ *as the set of fluent literals. A fluent formula* F *is any expression that can be constructed using fluent constants and the operators* $\neg, \wedge, \vee$ *(*$\rightarrow, \leftarrow, \leftrightarrow$ *can also be used for convenience); in addition we assume that true and false are special fluent formulae. For any* F, α, τ *and* τ', *the atoms* $\mathbf{Holds}(F, \tau)$, $\mathbf{Happens}(\alpha, \tau)$, $\tau \leq \tau'$ *and* $\mathbf{Initially}(F)$ *are allowed general atoms.*

Definition 7.2.3 ($\mathcal{ER}$**-formulae)** $\mathcal{ER}$ *formulae based on* Σ *are:*

- **direct effect rules** *of the form*

  $$a \textbf{ causes } l \textbf{ if } F'$$

  representing that l *becomes true whenever* a *is executed at a time when* F' *holds;*

- **derived effect rules** *of the form*

  $$\textbf{initiating } F \textbf{ causes } l \textbf{ if } F'$$

  representing that l *becomes true whenever* F *changes to true at a time when* F' *holds;*[5]

[4] In addition, we assume an equality predicate for $\mathcal{A}$ and $\mathcal{T}$ and we assume $\mathbf{t}, \mathbf{f} \in \mathcal{P}$

[5] We will call a or F the body, l the head and F' the condition of an effect rule.

- *any sentence constructed in the usual way of* **Holds**, **Happens**, $\leq$ *and* **Initially** *atoms and the connectives and quantifiers* $\neg, \wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \forall$ *and* $\exists$.

Some classes of sentences are of particular importance:

- **state constraints** of the form

$$\forall T : \mathbf{Holds}(F, T)$$

- **action preconditions** of the form

$$\forall T : \mathbf{Happens}(a, T) \rightarrow \mathbf{Holds}(F, T)$$

Other sentences may state complete or incomplete information about the truth of fluents at certain times, the occurrence and order of actions, or the initial state.

Definition 7.2.4 **($\mathcal{ER}$-theory)** *An $\mathcal{ER}$-theory is a tuple* $< \Sigma, \Pi_e, \Pi_p >$ *such that* Σ *is an $\mathcal{ER}$-signature,* Π_e *is a set of direct or derived effect rules based on* Σ, Π_p *is a set of sentences based on* Σ.

Now, let us explain and motivate the types of formulae proposed above. Direct effect rules are necessary constructs in even the simplest theories of action. They represent the simple immediate effects of all actions. Additional constructs are required when more complex issues are to be dealt with. Action preconditions for example are important when addressing the qualification problem, an issue to which we return later. They represent necessary and sufficient conditions that must be satisfied for an action to be able to occur.

Most importantly, we require constructs for addressing the ramification problem. A straightforward example of a ramification (i.e. an indirect effect) is that applying momentum to a gear wheel, which has the direct effect that the wheel starts turning, can also have the indirect effect that other wheels connected to it start turning. We need constructs allowing to correctly determine the complete set of direct and indirect effects an action gives rise to.

In the literature, ramifications have generally been considered strongly related to state constraints. In fact, the ramification problem has sometimes been defined as dealing with indirect effects *due to* state constraints. We prefer not to restrict ourselves to this subset, as we will argue that other indirect effects are just as important in practice. A state constraint is a fixed relation between fluents that needs to hold at all times. For example,

if two gear wheels are connected, a state constraint is that at any time point they must be either both turning or both stationary. This relates to indirect effects as follows: if an action's direct effects result in a violation of a state constraint (like making one gear wheel turn in the above example), this may give rise to indirect effects restoring the validity of the constraint (e.g. the other gear wheel will start turning as well).

Evidently, state constraints should not always give rise to ramifications. For example, suppose a person can be a manager only if he/she has a PhD. Presumably, we do not want a person to get a PhD as a side effect of being promoted. Rather we intend promotion to be prohibited for a person who has no PhD. In this case, the state constraint imposes an implicit precondition on the promotion. So, state constraints play an important role both as preconditions of actions and as causes of indirect effects. However, they are not sufficient for dealing with either problem.

It has been argued convincingly in for example [70] that state constraints are insufficient to convey all of the information required to determine valid sets of effects: it is unclear in general if an action violating a state constraint will give rise to indirect effects or if this action is simply impossible ([41]). Also, as argued in [40], [64] and [84], there can be multiple sets of indirect effects able to restore the validity of a state constraint, and the intended set cannot be determined without additional information. There exists in other words no automated, domain-independent method to derive the ramifications and qualifications corresponding to an arbitrary set of state constraints. To cure this problem, *causal laws* in some form or other have been proposed ([70, 50, 109, 42, 65]) to represent ramifications. Causal laws are explicit rules describing that certain changes in fluents cause (or may cause) certain other changes: an example would be a rule stating that making one gear wheel turn results in the turning of the other wheel. Thus causal laws provide a direct characterisation of the possible ramifications.

Interestingly, in all existing approaches incorporating them, causal laws are still tightly coupled with state constraints: in all aforementioned approaches they are used as a way of restoring integrity of some explicit or implicit state constraint.[6] We argue that this is an unnecessary and undesirable limitation: there is no reason why indirect effects should always correspond to a state constraint, as state constraints are not the "cause" of ramifications. In our view ramifications are simply manifestations of effect propagations. They represent some physical or logical force causing particular effects when certain other effects occur: for example when one gear wheel starts turning, it will give rise to physical forces making other

[6] We discuss the above approaches in more detail in section 7.10.

wheels turn as well.[7] State constraints, like in the above example that both wheels are turning or both are stationary, arise often as a *consequence* of particular effect propagation patterns. This does not in any way diminish the importance of state constraints, as they capture in a very concise and natural way a lot of information about a particular domain. However, it is wrong to assume the whole domain revolves around them: there is a lot going on which is not reflected in state constraints, these are only one high-level manifestation of the underlying mechanisms.

Consider the example of an alarm system that detects if somehow people enter a building. We assume the building has many possible entrances (doors, windows, possibly unexpected ways of getting in). So, there are many actions able to bring someone in the building and these actions may not even all be known.[8] We formalise the system using the fluents *in* (stating that there is someone inside), *active* (the alarm system is active) and *ring* (the alarm bell is ringing). While the system is active, anyone entering the building triggers the alarm: if *in* becomes true when *active* is already true, *ring* becomes true. In $\mathcal{ER}$ this reads

$$\textbf{initiating } \mathit{in} \textbf{ causes } \mathit{ring} \textbf{ if } \mathit{active}$$

However, the corresponding state constraint $\forall T : \mathbf{Holds}(\mathit{in} \wedge \mathit{active} \rightarrow \mathit{ring}, T)$ is not valid, since activating the alarm system when someone is already in the building is not supposed to cause the bell to ring. Moreover we can assume that the proposed constraint may also be violated by shutting down the bell, without deactivating the alarm system. So, there is no state constraint related to this triggered effect, it is simply caused by a different change.

A maybe even more obvious example of a system incorporating indirect effects unrelated to state constraints, is a simple electronic counter. Let us say the events it counts are represented by a transition of a certain voltage from low to high. This could be represented by rules like

$$\textbf{initiating } \mathit{volt}_1 \textbf{ causes } \mathit{count}(n+1) \textbf{ if } \mathit{count}(n)$$
$$\textbf{initiating } \mathit{volt}_1 \textbf{ causes } \neg\mathit{count}(n) \textbf{ if } \mathit{count}(n)$$

Also this indirect effect is in no way related to a state constraint. What is going on is just a propagation of effects, one effect triggering another one,

[7] However, the propagation is not necessarily a physical one: for example someone who stops being alive also starts being dead, which can be considered a "logical" effect propagation.

[8] Hence the use of an explicit rule relating the alarm bell to someone's presence in the building cannot be circumvented by adding new direct effect rules for each action (which would be an undesirable approach in any decent knowledge representation system in any case).

The above kinds of indirect effects cannot be correctly modelled by the existing approaches for dealing with ramifications. Therefore, we propose the following approach. We use independent effect propagation rules for representing ramifications. These look like the causal laws used in the literature, but differ essentially from them in that their semantics is independent of any state constraints in the theory (as opposed to the semantics of causal rules in [109]), and in that they do not include an implicit state constraint themselves (as opposed to causal laws in [65], [70], [50], and [42]). Thanks to this uncoupling of state constraints and derived effect rules, a wider range of indirect effects, including those in the above examples, can be modelled.

Given the direct and derived effect rules in an $\mathcal{ER}$-theory and a particular action or set of actions executed in a certain state, the resulting state after this action or set of actions is uniquely determined: actions produce some changes, which may lead to more changes by propagation through the derived effect rules. The role of state constraints is then reduced to filtering out models violating any state constraint in any state, which results in implicit action preconditions: if a state constraint would be violated by the combined direct and indirect effects of an action executed in a particular state, any interpretation in which that action occurs is not a model of the theory. Hence the action is impossible in that state. [9] Given a particular state constraint, whether or not a state violating the constraint will arise as a result of a particular action occurrence depends on the presence or absence in the theory of derived effect rules related to the constraint (which may restore its validity). This "restorability" can in turn be dependent on how the constraint was violated in the first place. For example, we know a dead turkey cannot be walking:

$$\forall T : \mathbf{Holds}(walking \rightarrow alive, T)$$

On the one hand, when a walking turkey dies, we know it will also stop walking. This is modelled by the derived effect rule :

$$\textbf{initiating } \neg alive \textbf{ causes } \neg walking \textbf{ if } true$$

On the other hand, one cannot resurrect a dead turkey by making it walk, so the rule **initiating** *walking* **causes** *alive* **if** *true* is not intended. In the absence of this rule, an action which makes *walking* true violates the state constraint if *alive* is false. Hence such an action, for example

[9] [66] describes an automated technique which, for a given set of direct effect rules of actions, derives from an arbitrary state constraint an equivalent explicit action precondition axiom. However, this technique does not take indirect effects into account.

start_walk, is then impossible: the state constraint functions as an implicit precondition.[10]

Apart from state constraints, $\mathcal{ER}$ also includes explicit action preconditions for dealing with qualifications. This is necessary because like indirect effects, also action preconditions are not necessarily related to state constraints. For example, in a chess game a move is only possible if the moved piece is initially on the starting position of the move. This cannot be represented by a state constraint, hence explicit preconditions are required. On the other hand, it is also undesirable to omit state constraints altogether, as they provide a very concise and natural way of representing information. To take the chess example again, a state constraint is that the player who has just made a move may not be in check. Compiling this constraint into a set of explicit move preconditions is certainly not the way to go.

The above discussion motivates the presence of state constraints, preconditions and derived effect rules in $\mathcal{ER}$. Now we still need to motivate the form of the derived effect rules, in particular why we need complex fluent formulae in the body of these rules. The first reason is conciseness: as we will illustrate, complex derived effect rules offer a very concise and natural way of representing indirect effects of actions. Strongly related to this is the observation that such rules, since they are triggered by combinations of effects, are perfectly suited for dealing with simultaneous actions. The issue of simultaneous actions will be discussed in section 7.4. Here we show the general applicability of complex derived effect rules.

As an example we present the suitcase domain from [65]. A suitcase is equipped with a spring mechanism which opens the suitcase when its two latches are open at the same time. This can happen in several different ways: both latches may be opened simultaneously, or one latch may already be open when the second one is opened. In the latter case, we need to ensure that the open latch is not just closed as the closed one is opened. Using a complex fluent formula, this set of possibilities can be represented by one derived effect rule

$$\textbf{initiating } l_1 \wedge l_2 \textbf{ causes } \mathit{open} \textbf{ if } \mathit{true}$$

where l_1, l_2 represent that latch 1 resp. 2 are open and *open* that the suitcase is open. Without complex fluent formulae, at least three rules would be needed, and they would need to be able to represent explicit

[10] [109] describes an automatic way of deriving the intended causal rules related to a state constraint, using additional *influence information*. For derived effect rules corresponding to state constraints, we can use a variant of this method in $\mathcal{ER}$. We discuss this in section 7.8.

absences of initiations.[11]

With action preconditions, complex effect rules, and state constraints, we are able to characterise the general laws ruling a temporal domain. Apart from that, we need to represent scenario information in such a domain, like actual action occurrences, an initial state, known fluent values at certain time points. We choose to represent these as a standard first order theory, as a part of Π_p: first of all because this is the simplest approach, and second because first order logic allows us to deal easily with incomplete scenario information, as we have discussed in Chapters 1 and 2. We illustrate this in the setting of $\mathcal{ER}$ in section 7.4.

7.3 The Semantics of $\mathcal{ER}$

In this section we discuss and define the semantics of $\mathcal{ER}$. As indicated, the most important concern is solving the frame and ramification problems. This is usually done by means of an inertia axiom stating that fluents persist unless they are changed, in combination with a representation of the closed world assumption, also described as a "minimisation of change". This can be achieved by using a circumscription policy or techniques extending Clark completion.

It is unclear to us if a variant of circumscriptive minimisation can yield a general solution to the frame and ramification problems. The many increasingly complex variants proposed to date suggest that such a general solution is not evident, even though distinct variants yield solutions for particular restricted classes of theories. The reason for these problems is in our view that the idea that change has to be minimised in some way is only an approximation of the "inertia" we observe in the real world. It does not entirely correspond to our intuition.

Clark completion in turn formalises a simple and intuitive principle ("a change occurs if and only if we say so"), but is only applicable to a very restricted class of theories (i.e. theories without recursion in the effect rules). On the other hand, as we have discussed in Chapter 2, more powerful extensions of Clark completion are the more advanced logic programming semantics like stable, well-founded or justification semantics. We will apply such an extension of Clark completion to $\mathcal{ER}$'s effect rules.

The semantics we propose for a set of effect rules is to read them as an inductive definition of a predicate *causes*. Provided there are no cycles in the effect rules, this simply coincides with their completion. However,

[11] Lin also utilises complex causal laws in [65]. However, these laws differ from our derived effect rules in that they incorporate a state constraint component.

we argue that cyclic dependencies naturally occur in effect rules, and that there is a natural way to deal with them.[12]

Consider two connected gear wheels. Any action which makes one gear turn, makes the other one turn as well, and any action which stops one gear, stops the other one. This can be represented by the following rules:

initiating *turning_1* **causes** *turning_2* **if** *true*
initiating $\neg$*turning_1* **causes** $\neg$*turning_2* **if** *true*
initiating *turning_2* **causes** *turning_1* **if** *true*
initiating $\neg$*turning_2* **causes** $\neg$*turning_1* **if** *true*

which introduce a cyclic dependency, though the example is certainly not far-fetched or unnatural. Evidently, given such mutually dependent effects, no effect should take place unless some exterior effect causes it (e.g. in the example a motor is started). In other words we comply with Shoham ([104]) who insists that causation is anti-reflexive, i.e. that causes for a fact should never include the fact itself. However, we claim that this condition should not be enforced by ruling out cycles in the causal rules on a syntactic level: the example shows that such cycles naturally arise in quite normal problem domains. Rather, cyclic dependencies should be given their intuitive meaning, which is that if and only if one of the mutually dependent effects has an "external cause", both of them occur.

Negative dependencies (in the sense that the absence of a particular effect is a precondition for another effect to occur) do at first sight not occur in effect rules, but a closer look at the complex effect rules reveals that this is a false impression. Take the suitcase domain presented earlier, which contains the rule

initiating $l_1 \wedge l_2$ **causes** *open* **if** *true*.

The intended reading of this rule is that *open* is initiated if the conjunction $l_1 \wedge l_2$ becomes true. This can happen in three ways, intuitively

if l_1 is initiated and l_2 is initiated
if l_1 is initiated, l_2 holds and $\neg l_2$ is not initiated
if l_2 is initiated, l_1 holds and $\neg l_1$ is not initiated

In the last two cases, the initiation of $l_1 \wedge l_2$ depends on the absence of a primitive initiation, so there are negative dependencies. Below we will define the semantics of complex effect rules by mapping them to an equivalent set of primitive rules like those above and by interpreting these primitive

[12] The discussion below is hence also a motivation for the formalisations of the closed world assumption in (open) logic programming semantics.

rules as an inductive definition. These rules contain negative dependencies, yet giving them a natural semantics needs not be problematic: in the above example it is clear that when l_1 is initiated while l_2 holds, *open* is expected to be initiated rather than l_2 terminated[13]. In general, the intended semantics is clear if the effects can be ordered in layers (*stratified*) such that each effect only depends negatively on more primitive effects, i.e. effects in lower layers.[14]

As we do not impose syntactic constraints ensuring stratifiability of a theory, in some cases a set of rules can have an ambiguous meaning or be completely nonsensical. As an example of the first case, consider adding the rule

$$\textbf{initiating } l_1 \wedge \neg open \textbf{ causes } \neg l_2 \textbf{ if } true$$

to the above example. Now the initiations of *open* and $\neg l_2$ depend negatively on each other. This leads to the following problem: assume $\neg open$, $\neg l_1$ and l_2 hold at a certain time, i.e. the suitcase is closed but one latch is open. Then the other latch is opened, i.e. l_1 is initiated. If now $\neg open$ would persist then according to the newly introduced rule $\neg l_2$ would be initiated. But if l_2 would persist, the original rule **initiating** $l_1 \wedge l_2$ **causes** *open* **if** *true* would force *open* to be initiated. Both possibilities are in accordance with the rules, and no non-ambiguous conclusion is possible. In this case one can at best argue that the effect is nondeterministic, but in our view nondeterminism, if intended, should be modelled explicitly and not follow from tricky combinations of rules which themselves do not hint at nondeterminism. We will introduce explicit nondeterministic rules in our theory in a later section.

The other case of a non-stratified definition, in which an effect negatively depends on itself, i.e. it occurs provided it does not occur, is clearly nonsensical. We handle both cases of non-stratified definitions by assigning an "undefined" truth value to effects that depend negatively on effects in the same layer. This truth value is interpreted as indicating an error in the definition, in the sense that the definition is not constructive. By dealing with non-constructive definitions in the indicated way, we avoid introducing complex syntactic restrictions ensuring stratification, and keep our approach general.

The above intuitions are formalised by the principle of inductive definition. This principle is well-suited for representing effect propagations due to its constructiveness: the truth of atoms propagates through definition rules like effects propagate due to physical or logical forces. Hence no atom

[13] We say a fluent f is terminated iff $\neg f$ is initiated.

[14] In the example, *open* is in a higher layer than l_1 and l_2.

can be true without a cause and a cause for a true atom can never depend on the atom itself.

7.3.1 Principle of Inductive Definition

The semantics and expressiveness of inductive definitions are studied in a sub-area of mathematical logic, the area of Iterated Inductive Definitions (IID) ([16, 81, 1]). We formalise this semantics in a different way and extend it to non-stratified definitions.

We need the following concepts.

Definition 7.3.1 ($\mathcal{V}_{\mathbf{P}}$,$\leq_F$) *Given a set of ground atoms* $\mathbf{P}$*, the set* $\mathcal{V}_{\mathbf{P}}$ *of (3-valued) valuations on* $\mathbf{P}$ *is the set of all functions* $\mathbf{P} \rightarrow \{\mathbf{t}, \mathbf{u}, \mathbf{f}\}$. *On* $\mathcal{V}_{\mathbf{P}}$*, a partial order* $\leq_F$ *is defined as the pointwise extension of the order* $\mathbf{u} \leq_F \mathbf{t}, \mathbf{u} \leq_F \mathbf{f}$*; more precisely,* $\forall I, I' \in \mathcal{V}_{\mathbf{P}} : I \leq_F I'$ *iff* $\forall l \in \mathbf{P} : I(l) \leq_F I(l')$.

It is easy to prove that $\mathcal{V}_{\mathbf{P}}, \leq_F$ is a chain complete poset[15] with least element $\perp$, the valuation which assigns $\mathbf{u}$ to each atom.

Definition 7.3.2 (inductive definition) *Given a set of ground atoms* $\mathbf{P}$*, we define* $\widehat{\mathbf{P}} = \mathbf{P} \cup \{\neg l | l \in \mathbf{P}\} \cup \{\mathbf{t}, \mathbf{f}\}$.[16] *Valuations can be naturally extended to* $\widehat{\mathbf{P}}$. *A* definition rule *in* $\mathbf{P}$ *is an object* $l \leftarrow B$ *where* $l \in \mathbf{P}$ *and* $B \subseteq \widehat{\mathbf{P}}$. l *is called the head,* B *the body of the rule. A definition on* $\mathbf{P}$ *is any set* $\mathcal{D}$ *of rules in* $\mathbf{P}$.

Given $\mathbf{P}$ and a definition $\mathcal{D}$ on $\mathbf{P}$, we need to characterise a valuation $I_{\mathcal{D}}$ which defines the truth values of all atoms according to $\mathcal{D}$. In IID this involves stratifying the definition, but it has been argued in [24] that techniques inspired by logic programming semantics formalise the same intuitions in a more general and syntax independent way. We present this technique.

Definition 7.3.3 (proof tree) *A proof tree* T *for an atom* $p \in \mathbf{P}$ *is a tree of elements of* $\widehat{\mathbf{P}}$ *such that*

- *the root of* T *is* p
- *for each non-leaf node* n *of* T *with immediate descendants* B*, "*$n \leftarrow B$*"* $\in \mathcal{D}$ *or* $B = \{\mathbf{f}\}$ *(Hence, each atom has at least one (false) proof tree.)*

[15] For more details on particular sorts of partially ordered structures, as well as on operators on these structures and fixpoints, which we will use below, we refer to [7], [69] and [61].

[16] $\mathbf{u}$ should never occur explicitly in a definition.

- *T is maximal, i.e. atoms occur only in non-leaf nodes. Leaf nodes then contain only* **t**, **f** *or a negative literal.*
- *T is finite, i.e. contains no infinite branches*

Given some valuation $I \in \mathcal{V}_{\mathbf{P}}$ of $\mathbf{P}$, for each $l \in \mathbf{P}$, we define its *supported value* w.r.t. I, denoted $SV_I(l)$, as the truth value proven by its "best" proof tree. Formally:

Definition 7.3.4 (supported value)

- $SV_I(l) = \mathbf{t}$ *if l has a proof tree with all leaves containing true facts w.r.t. I;*
- $SV_I(l) = \mathbf{f}$ *if each proof tree of l has a false fact w.r.t. I in a leaf;*
- $SV_I(l) = \mathbf{u}$ *otherwise; i.e. if each proof tree of l contains a non-true leaf, and some proof tree contains only non-false leaves.*

For a definite definition $\mathcal{D}$, $I_{\mathcal{D}}$ is the valuation mapping each $p \in \mathbf{P}$ to $SV_{\perp}(p)$, i.e. each atom is mapped to its supported value (w.r.t. $\perp$). For non-definite definitions, $I_{\mathcal{D}}$ is obtained as a fixpoint of this operation:

Definition 7.3.5 ($\mathcal{PI_D}$) *The* positive induction operator $\mathcal{PI_D} : \mathcal{V}_{\mathbf{P}} \rightarrow \mathcal{V}_{\mathbf{P}} : I \rightarrow I'$ *is defined such that* $\forall p \in \mathbf{P} : I'(p) = SV_I(p)$.

It can be proven that this operator is monotonic and hence always has a least fixpoint $\mathcal{PI_D}\!\uparrow$. This allows us to define $I_{\mathcal{D}}$ as:

Definition 7.3.6 *Given* $< \mathbf{P}, \mathcal{D} >$, $I_{\mathcal{D}} = \mathcal{PI_D}\!\uparrow$.

7.3.2 Tackling the Ramification Problem

We now show how to handle derived effect rules using an inductive definition based semantics. We present the approach in a language-independent way, such that it can be embedded in most currently existing formalisms like Situation Calculus, Event Calculus or $\mathcal{A}$ style languages. We embed the approach in $\mathcal{ER}$ in the next subsection.

The problem we want to deal with is the following: given the state of the world at a certain instant in time, a set of actions occurring at that time and a set of direct and derived effect rules, calculate the state of the world resulting after these actions. We denote a set of actions as A and a state as St. The set of all states is written as $\mathcal{S}t$.

We reduce direct and derived effect rules to an inductive definition of **Init** and **Causes**. These predicates intuitively denote strong and weak initiation, respectively: **Init**(A, St, l) means that l does not hold in St but holds in the successor state resulting after the application of the set of (simultaneous) actions A in St. **Causes**(A, St, l) means that l holds in the successor state of St after A, but possibly also already in St. The fluent literals l that are true in the successor state are the ones that are initiated and those that are true in St and of which the negation $\bar{l}$ is not initiated.

The intended reading of an effect rule **initiating** F **causes** l **if** F' is that given F', the change in truth value of F from false to true (strong initiation of F) causes l to become true if it was not already true (i.e. weakly initiates l). To formalise this for complex F we introduce the concept of a supporting set. This concept is based on a disjunctive normal form of the formula. Since a definition can have 3-valued interpretations, this normal form needs to be equivalence preserving under 3-valued FOL semantics. The 2-valued disjunctive normal form does not satisfy this property (since for example $F \wedge \neg F$ is not 3-valued equivalent to $\mathbf{f}$), but it is easy to derive a 3-valued variant:

Definition 7.3.7 (3-valued disjunctive normal form) *The 3-valued disjunctive normal form $3dnf(F)$ of a fluent formula F is obtained by applying the following rewriting rules to F or its constituents (if $\leftrightarrow$, $\rightarrow$ or $\leftarrow$ occur in F we assume they are rewritten in terms of $\neg$, $\wedge$, $\vee$ as usual) until no further rules apply:*[17]

- *replace* $\neg\neg F$ *by* F
- *replace* $\neg(F \wedge G)$ *by* $\neg F \vee \neg G$
- *replace* $\neg(F \vee G)$ *by* $\neg F \wedge \neg G$
- *replace* $F \wedge (G \vee H)$ *by* $(F \wedge G) \vee (F \wedge H)$
- *replace* $F \wedge F$ *by* F
- *replace* $F \vee (F \wedge G)$ *by* F

The following properties can be proven. The rewriting process always terminates (since each step moves a $\neg$ symbol inward, moves a $\wedge$ symbol inward without moving a $\neg$ outward, or eliminates $\neg$, $\wedge$ or $\vee$ symbols without moving $\wedge$ or $\neg$ outward). The resulting normal form $3dnf(F)$ is unique[18] i.e. independent of the order in which subformulae are processed (this is

[17] We assume commutativity and associativity are applied whenever needed.

[18] modulo commutativity and associativity

easily verified by case analysis). $3dnf(F)$ is always a disjunction of conjunctions of literals (since otherwise one of the four first rules applies). $3dnf(F)$ is equivalent to F under 3-valued as well as 2-valued semantics (since all rewriting rules are equivalence preserving under either semantics). Finally, under either semantics, if F and G are equivalent then so are $3dnf(F)$ and $3dnf(G)$ (this follows immediately from the previous property).

We define the concept of supporting set as follows:

Definition 7.3.8 (supporting set) *Let F be a fluent formula and $F' = (l_1^1 \wedge .. \wedge l_{n_1}^1) \vee ... \vee (l_1^m \wedge .. \wedge l_{n_m}^m)$ its 3-valued disjunctive normal form. A supporting set L of F is any set $\{l_1^i, .., l_{n_i}^i\}, \forall 1 \leq i \leq m$.*

A formula is true if and only if all literals of some supporting set of it are true. It follows that F is *initiated* iff F is not already true and for some supporting set L of F, all literals of some $L_i \subseteq L$ are initiated and all literals in $L_p = L \setminus L_i$ are true and not terminated. This leads to the formalisation below.

Definition 7.3.9 (notations)
In what follows $Ha(a, A)$ is the truth value of "$a \in A$", $Ho(l, St)$ is the truth value of "$l \in St$" and $Ho(F, St)$ for complex formulae is the truth value obtained from $Ho(l, St)$ for literals in the classical way. We define **Init**(A, St, L) *as* $\{$**Init**$(A, St, l) \mid l \in L\}$ *and* **Causes**(A, St, L) *as* $\{$**Causes**$(A, St, l) \mid l \in L\}$. *Further, from now on we use the notation $\overline{f} = \neg f$, $\overline{\neg f} = f$, $\overline{L} = \{\overline{l} \mid l \in L\}$ for any set of literals L, and $\overline{P} = \{\neg p \mid p \in P\}$ for any set of* **Init** *or* **Causes** *atoms P.*

Definition 7.3.10 (definition induced by effect rules)
The definition induced by a rule "a **causes** *l* **if** *F" is*[19]

$$\{\mathbf{Causes}(A, St, l) \leftarrow Ha(a, A), Ho(F, St) \mid A \subseteq \mathcal{A},\ St \in \mathcal{S}t\}.$$

*The definition induced by a rule "***initiating** *F* **causes** *l* **if** *F'" is*

$$\{\mathbf{Causes}(A, St, l) \leftarrow \mathbf{Init}(A, St, L_i), \overline{\mathbf{Init}(A, St, \overline{L_p})}, Ho(L_p, St), \neg Ho(F, St), Ho(F', St)$$
$$\mid A \subseteq \mathcal{A},\ St \in \mathcal{S}t \text{ and } L_i \cup L_p \text{ is a supporting set of } F\}.$$

We define the definition induced by a set of effect rules Π_e as $\mathcal{D}_{init} = \mathcal{D}_g \cup$ $\{$**Init**$(A, St, l) \leftarrow$ **Causes**$(A, St, l), \neg Ho(l, St) \mid A \subseteq \mathcal{A},\ St \in \mathcal{S}t, l \in \hat{\mathbf{P}}\}$, *where $\mathcal{D}_g$ is the union of the definitions induced by all rules in Π_e.*

[19] Recall that $Ha(a, A)$ and $Ho(F, St)$ are the *truth values* of "$a \in A$" and "$l \in St$", not the formulae themselves.

$\mathcal{D}_{init}$ is an inductive definition on the atom domain $\mathbf{P}' = \{\mathbf{Init}(A, St, l), \mathbf{Causes}(A, St, l) \mid A \subseteq \mathcal{A},\ St \in \mathcal{S}t, l \in \widehat{\mathbf{F}}\}$, for which $I_{\mathcal{D}_{init}}$ is the least fixpoint of $\mathcal{PI}_{\mathcal{D}_{init}}$.

The rules $\mathbf{Init}(A, St, l) \leftarrow \mathbf{Causes}(A, St, l), \neg Ho(l, St)$ are the only rules for **Init**. Since the completion of the definition rules is entailed by the inductive definition semantics, the rules imply

$$\forall St, A, l : [\mathbf{Init}(A, St, l) \leftrightarrow \mathbf{Causes}(A, St, l) \wedge \neg Ho(l, St)]$$

and thereby capture the intended relation between strong and weak initiation.

The mutual recursion in the definitions of **Init** and **Causes** indicates that strong initiations may provide causes for literals to become true; only if these literals were not already true, i.e. if they were actually changed, they can themselves give rise to further ramifications. We have used the predicates **Causes** and **Init** to stress this important distinction between strong and weak initiation. However, given the above relation between **Causes** and **Init**, we could get rid of the **Init** predicate using the following theorem:

Theorem 7.3.1
Given that $\forall St, A, l : [\mathbf{Init}(A, St, l) \leftrightarrow \mathbf{Causes}(A, St, l) \wedge \neg Ho(l, St)]$, *the definition* $D_1 =$

$$\{\mathbf{Causes}(A, St, l) \leftarrow \mathbf{Init}(A, St, L_1), \overline{\mathbf{Init}(A, St, \overline{L_2})}, Ho(L_2, St), \neg Ho(F, St), Ho(F', St)$$
$$\mid A \subseteq \mathcal{A},\ St \in \mathcal{S}t \textit{ and } L_1 \cup L_2 \textit{ is a supporting set of } F\}.$$

is equivalent to $D_2 =$

$$\{\mathbf{Causes}(A, St, l) \leftarrow \mathbf{Causes}(A, St, L_1), Ho(\overline{L_1}, St), \overline{\mathbf{Causes}(A, St, \overline{L_2})}, Ho(L_2, St), \neg Ho(F, St), Ho(F', St)$$
$$\mid A \subseteq \mathcal{A},\ St \in \mathcal{S}t \textit{ and } L_1 \cup L_2 \textit{ is a supporting set of } F\}.$$

Intuitively, this is true because if at least one rule body for a particular literal is true in D_1 in a particular state, then at least one rule body is true for the same literal in D_2 in the same state, and vice versa.

Proof:
We first prove the first part: assume a rule in D_1 with a particular L_1 and L_2 and which has a true body in St. $L_1 \cup L_2$ is a supporting set of F. For the literals in L_1, it holds that $\mathbf{Init}(A, St, l)$, hence $\mathbf{Causes}(A, St, l) \wedge \neg Ho(l, St)$, which entails $\mathbf{Causes}(A, St, l)$. For the literals in L_2 it holds that $Ho(l, St) \wedge \neg\mathbf{Init}(A, St, \bar{l})$, hence $(Ho(l, St) \wedge \neg\mathbf{Causes}(A, St, \bar{l})) \vee (Ho(l, St) \wedge Ho(\bar{l}, St))$, which is equivalent to $(Ho(l, St) \wedge \neg\mathbf{Causes}(A, St, \bar{l}))$.

Combining these results, we find that the rule in D_2 with the same L_1 and L_2 has a true body.

On the other hand, assume a rule in D_2 with particular L_1 and L_2 and which has a true body in a particular St. For the literals in L_1, $\mathbf{Causes}(A, St, l)$ holds. We can partition L_1 into a set of literals which are true in St (L_{1a}) and a set of literals which are false in St (L_{1b}).[20] For the literals in L_{1b}, $\mathbf{Causes}(A, St, l) \land \neg Ho(l, St)$ is true, which is equivalent to $\mathbf{Init}(A, St, l)$. For the literals in L_{1a}, $\mathbf{Causes}(A, St, l) \land Ho(l, t)$ is true, which implies $\neg\mathbf{Causes}(A, St, \bar{l}) \land Ho(l, St)$. For the literals in L_2, $Ho(l, St) \land \neg\mathbf{Causes}(A, St, \bar{l})$ holds, which implies $Ho(l, St) \land \neg\mathbf{Init}(A, St, \bar{l})$. Combining these results, we find that a D_1-rule constructed with L_{1b} as the new L_1' and $L_2 \cup L_{1a}$ as the new L_2' has a true body, with $L_1' \cup L_2' = L_1 \cup L_2$ the same supporting set. □

Despite this equivalence result, we prefer to use both the **Init** and **Causes** predicates explicitly for reasons of clarity.

We now give some interesting results concerning the proposed semantics.

Theorem 7.3.2 *Given a state St, a set of actions A and a set of direct and derived effect rules, the truth values of* $\mathbf{Init}(A, St, l)$ *for all l are uniquely determined.*

Proof:

This can be proven as follows: the definition induced by a specific set of effect rules is a definition in which (by its construction) the set of rules for $\mathbf{Init}(A, St, l)$ and $\mathbf{Causes}(A, St, l)$ for each l depends only on the truth values $Ha(a, A)$ and $Ho(F, St)$. Given St and A, these truth values are uniquely determined. Hence, the set of rules for all atoms of the form $\mathbf{Init}(A, St, l)$ or $\mathbf{Causes}(A, St, l)$ is unique. Moreover, in the entire set of rules for a given St and A, the supported values of $\mathbf{Init}(A, St, l)$ and $\mathbf{Causes}(A, St, l)$ only depend directly or indirectly on the truth values of other atoms of the form $\mathbf{Init}(A, St, l')$ and $\mathbf{Causes}(A, St, l')$ for the same St and A. Hence, the set of rules for each particular St and A is a separate definition determining the values of all atoms in the domain $\{\mathbf{Init}(A, St, l), \mathbf{Causes}(A, St, l) \mid l \in \widehat{\mathrm{P}}\}$ independent of all other states or actions. $I_{D_{init}}$ for this definition is unique, which proves the theorem. □

The theorem guarantees that successor states generated by a set of deterministic effect rules are always unique. In this respect our approach differs from the one in [109]. Unless nondeterminism is explicitly introduced, the theory leaves no room for ambiguity.

[20] L_{1b} is not empty: if it were, all literals in the supporting set $L_1 \cup L_2$ would already be true, which is in contradiction with the condition in the rule body that F should not hold.

The following results give some alternative characterisations of the above semantics in several special cases. Moreover they shed some light on the relation to existing approaches to the frame and ramification problems.

Definition 7.3.11 (fluent dependency)
We say a fluent f occurring in **initiating** F **causes** f **if** F' *or in* **initiating** F **causes** $\neg f$ **if** F' *depends on a fluent f' if f' occurs in F, or if a fluent which depends on f' occurs in F.*

Theorem 7.3.3 *If the derived effect rules are acyclic, i.e. if no fluent depends on itself, $I_{\mathcal{D}_{init}}$ is always 2-valued. Moreover $I_{\mathcal{D}_{init}}$ coincides with the unique model of the Clark completion of the definition rules.*

Proof:
The rules in the definition induced by a rule **initiating** F **causes** l **if** F' contain only literals $\mathbf{Init}(A, St, l')$ or $\neg\mathbf{Init}(A, St, l')$ for l' which are fluent literals occurring in F or negations of such fluent literals. The dependency relation is therefore the same for fluents in the effect rules as for literals containing them in the grounding. Hence, if the effect rules are acyclic, then so are the definition rules in the grounding. As we mentioned before, the positive induction operator has the same fixpoints as the well-founded operator, so an inductive definition is formally equivalent with a logic program under well-founded semantics. Under this reading, acyclic definitions correspond to acyclic logic programs, for which the well-founded semantics has been proven to be 2-valued and to coincide with the completion semantics. The theorem follows immediately. □

Theorem 7.3.4 *If the body of each derived effect rule is a single literal, $I_{\mathcal{D}_{init}}$ is always 2-valued, and coincides with the unique model of the parallel circumscription*[21] *of* **Init** *and* **Causes** *in the theory consisting of the definition rules read as implications.*

Proof:
If the derived effect rules have only single literals as bodies, the rules in $\mathcal{D}_{init}$ contain no negative literals. Hence, they are equivalent to a definite logic program, for which the well-founded semantics has been proven to coincide with the perfect model semantics of [86] in [116]. The perfect model semantics is 2-valued, which proves the first result. Note moreover that $I_{\mathcal{D}_{init}}$ is unique, so there is one unique perfect model.

Further, in [86] it is proven that for definite programs the perfect models coincide with the minimal models of the program read as a set of implications. On the other hand, in [62] it has been shown that an interpretation

[21] Parallel circumscription is circumscription on multiple predicates at the same time, without priorities.

is a model of $Circum(A; P; Z)$ (where A is the given theory, P the set of predicates to be circumscribed and Z a set of predicates allowed to vary) iff it is minimal in the class of models of A with respect to $\leq^{P;Z}$. $M_1 \leq^{P;Z} M_2$ iff M_1 and M_2 differ only in the predicates in $P \cup Z$ and the extension of each predicate of P in M_1 is a subset of its extension in M_2. In our case, A is the set of rules read as implications, Z is empty and P contains both **Init** and **Causes**, the only predicates occurring in the definition. Then $M_1 \leq^{P;Z} M_2$ iff the set of true atoms in M_1 is a subset of that in M_2. Hence, the minimal models with respect to $\leq^{P;Z}$ are the minimal models of the program read as a set of implications. It follows that for simple effect rules, the model $I_{\mathcal{D}_{init}}$ coincides with the unique perfect model, which in turn coincides with the model of the circumscription. This proves the theorem. □

The above results show that for several classes of definitions for which other semantics are known to assign the intended meaning to all predicates, the inductive definition semantics coincides with these semantics. However the inductive definition semantics is more general, also dealing with definitions that cannot be dealt with by the more common semantics.

7.3.3 Formal Semantics of $\mathcal{ER}$

We are now ready to define the semantics of $\mathcal{ER}$, embedding the semantics of effect rules given above in the specific $\mathcal{ER}$ setting.

Definition 7.3.12 (temporal interpretation) *Given an $\mathcal{ER}$-theory $\Pi = <\Sigma, \Pi_e, \Pi_p>$, a temporal interpretation of Π is a structure $I = <\mathbf{P}, \mathcal{F}un, \mathcal{H}>$ with:*

$\mathbf{P} = \{t_1 \leq t_2 \mid t_1, t_2 \in \mathbf{T}\} \cup$
$\quad \{\mathbf{Initially}(l) \mid l \in \widehat{\mathbf{F}}\} \cup$
$\quad \{\mathbf{Happens}(a, t) \mid a \in \mathbf{A}, t \in \mathbf{T}\} \cup$
$\quad \{\mathbf{Holds}(l, t) \mid l \in \widehat{\mathbf{F}}, t \in \mathbf{T}\} \cup$
$\quad \{\mathbf{Init}(t, l) \mid t \in \mathbf{T}, l \in \widehat{\mathbf{F}}\} \cup$
$\quad \{\mathbf{Causes}(t, l) \mid t \in \mathbf{T}, l \in \widehat{\mathbf{F}}\}$

$\mathcal{F}un : \mathbf{T} \to \mathbf{R}$, *a mapping of time constants to reals such that each real number is mapped to itself (recall that $\mathbf{R} \subseteq \mathbf{T}$)*

$\mathcal{H} : \mathbf{P} \to \{\mathbf{t}, \mathbf{f}\}$, *a valuation.*

$\mathcal{H}$ defines relations interpreting **Happens**, **Holds**, $\leq$, **Initially**, **Init** and **Causes**; we denote them $\mathcal{H}a, \mathcal{H}o, \preceq, \mathcal{I}nitially, \mathcal{I}nit, \mathcal{C}auses$ respectively.[22]

[22] Implicitly, $\mathbf{P}$ contains also all well-typed equality atoms and $\mathbf{t}, \mathbf{f}$. $\mathcal{H}$ defines their natural interpretation; in particular $=$ is interpreted as the identity. Moreover for all t, $\mathcal{H}o(true, t) = \mathcal{I}nitially(true) = \mathbf{t}$ and $\mathcal{H}o(false, t) = \mathcal{I}nitially(false) = \mathbf{f}$.

The predicates **Init** and **Causes**, which do not occur in Π_p, denote strong and weak initiation, respectively: $\mathbf{Init}(t, l)$ means that l does not hold at t but starts to hold immediately after t. $\mathbf{Causes}(t, l)$ means that l holds immediately after t, but can also hold at t. These predicates are determined by Π_e.

A temporal interpretation needs to satisfy the following conditions

- *$\preceq$ is the classical total order on* $\mathbf{R}$.
- *Well-founded event topology: the set* $\mathbf{E} = \{t | \exists a : \mathcal{H}a(a, t)\}$ *has a least element, denoted* e_{start} *and* $\forall t \preceq t' : [t, t'] \cap \mathbf{E}$ *is a finite set.*[23]
- *Consistency:* $\forall t, \forall f : \neg Causes(t, f) \vee \neg Causes(t, \neg f)$

  $\forall f : \mathcal{I}nitially(f) \leftrightarrow \neg \mathcal{I}nitially(\neg f)$

  These formulae denote initiation consistency and initial state consistency. Consistency ($\forall t, \forall f : \mathcal{H}o(f, t) \leftrightarrow \neg \mathcal{H}o(\neg f, t)$) follows from these two formulae and the inertia axiom given a well-founded event topology, as can be proven by induction on events.
- *Definition of initial state:* $\forall t \leq e_{start} : \mathcal{H}o(l, t) \leftrightarrow \mathcal{I}nitially(l)$;
- *Inertia:* $\forall t_1, t_2, \forall l :$

  $t_1 \prec t_2 \wedge (\neg \exists t_3 : t_1 \preceq t_3 \prec t_2 \wedge Causes(t_3, \bar{l}))$

  $\rightarrow \mathcal{H}o(l, t_2) \leftrightarrow Causes(t_1, l) \vee \mathcal{H}o(l, t_1)$

A temporal interpretation I is a model of an $\mathcal{ER}$-theory $< \Sigma, \Pi_e, \Pi_p >$ iff it is a model of both Π_e and Π_p. To define whether I satisfies Π_p, we extend the truth function $\mathcal{H}$ to all closed formulae F in the classical way. For complex **Holds** and **Initially** atoms, the interpretation of $\mathbf{Holds}(F, \tau)$ is defined as the interpretation of the formula F' obtained from F by substituting each fluent atom f by $\mathbf{Holds}(f, \mathcal{F}un(\tau))$. The interpretation of $\mathbf{Initially}(F)$ is defined likewise. An interpretation I satisfies Π_p iff all formulae in Π_p are true in I.

Next we focus on the semantics of the effect theory Π_e. This semantics is based on the inductive definition semantics for effect rules given above. At a particular time t, if the state St is the set of literals true at t, the truth value $Ho(F, St)$ corresponds to the truth value $\mathcal{H}o(F, t)$. Likewise, if A is the set of actions occurring at t, $Ha(a, A)$ corresponds to $\mathcal{H}a(a, t)$. $\mathbf{Init}(A, St, l)$ and $\mathbf{Causes}(A, St, l)$ then correspond to $\mathbf{Init}(t, l)$ and $\mathbf{Causes}(t, l)$.

[23] This condition plays the same role as the induction axiom in Chapter 5. The first part of the condition ensures that there is no sequence of events extending infinitely into the past. The second part is our *non-intermingling principle* ([36]) : it disallows an infinite number of actions (and hence, changes in truth value of a fluent) in a finite period of time. Together the two parts guarantee that each time point is only preceded by a finite number of actions.

Definition 7.3.13 (grounding)
The grounding of a direct effect rule "a **causes** *l* **if** *F" is* [24]

$\{\mathbf{Causes}(t,l) \leftarrow \mathcal{H}a(a,t), \mathcal{H}o(F,t) \mid t \in \mathbf{T}\}$.

*The grounding of a derived effect rule "***initiating** *F* **causes** *l* **if** *F′" is*

$$\{\mathbf{Causes}(t,l) \leftarrow \mathbf{Init}(t,L_i), \overline{\mathbf{Init}(t,\overline{L_p})}, \mathcal{H}o(L_p,t), \neg\mathcal{H}o(F,t), \mathcal{H}o(F',t) \mid t \in \mathbf{T} \text{ and } L_i \cup L_p \text{ is a supporting set of } F\}.$$

The grounding $\mathcal{D}_{init}$ *of a set of effect rules* Π_e *is* $\mathcal{D}_g \cup \{\mathbf{Init}(t,l) \leftarrow \mathbf{Causes}(t,l), \neg\mathcal{H}o(l,t) \mid t \in \mathbf{T}, l \in \widehat{\mathbf{P}}\}$, *where* $\mathcal{D}_g$ *is the union of the groundings of all rules in* Π_e.

$\mathcal{D}_{init}$ is an inductive definition on the atom domain $\mathbf{P}' = \{\mathbf{Init}(t,l), \mathbf{Causes}(t,l) \mid t \in \mathbf{T}, l \in \widehat{\mathbf{F}}\}$, for which $I_{\mathcal{D}_{init}}$ is defined as the least fixpoint of $\mathcal{PI}_{\mathcal{D}_{init}}$.

Taking everything together now, we obtain the following definition of a model of an $\mathcal{ER}$-theory.

Definition 7.3.14 ($\mathcal{ER}$-model)
Given an $\mathcal{ER}$*-theory* $\Pi_{\mathcal{ER}} = <\Sigma, \Pi_e, \Pi_p>$, *a temporal interpretation* I *is a model of* $\Pi_{\mathcal{ER}}$, *denoted* $I \models \Pi_{\mathcal{ER}}$, *iff* $I \models \Pi_p$ *and* $I \models \Pi_e$, *where*
$I \models \Pi_p$ *iff* $\forall F \in \Pi_p : \mathcal{H}(F) = \mathbf{t}$.
$I \models \Pi_e$ *iff* $\forall t \in \mathbf{T}, l \in \widehat{\mathbf{F}}$:
$\mathcal{I}nit(t,l) \leftrightarrow I_{\mathcal{D}_{init}}(\mathbf{Init}(t,l))$ *and* $\mathcal{C}auses(t,l) \leftrightarrow I_{\mathcal{D}_{init}}(\mathbf{Causes}(t,l))$.

Note that when $I_{\mathcal{D}_{init}}$ contains any truth value $\mathbf{u}$, the condition $I \models \Pi_e$ is unsatisfiable since $\mathcal{I}nit$ is a 2-valued relation. Recall that definitions with a 3-valued model are ambiguous (non-constructive) and are considered erroneous in our approach.

7.3.4 Properties of $\mathcal{ER}$

The following results carry over from the inductive definition semantics given before:

The rules $\mathbf{Init}(t,l) \leftarrow \mathbf{Causes}(t,l), \neg\mathcal{H}o(l,t)$ are the only rules for **Init**. Since the completion of the definition rules is entailed by the inductive definition semantics, the rules imply

$$\forall t, l : [\mathbf{Init}(t,l) \leftrightarrow \mathbf{Causes}(t,l) \wedge \neg\mathcal{H}o(l,t)]$$

We then find an immediate corollary of theorem 7.3.1:

[24] Observe that $\mathcal{H}a(a,t)$ and $\mathcal{H}o(F,t)$ are the *truth values* of "**Happens**(a,t)" and "**Holds**(F,t)".

Corollary 7.3.1 *Given that* $\forall t, l : [\mathbf{Init}(t,l) \leftrightarrow \mathbf{Causes}(t,l) \wedge \neg\mathcal{H}o(l,t)]$, *the definition* $D_1 =$

$$\{\mathbf{Causes}(t,l) \leftarrow \mathbf{Init}(t,L_1), \overline{\mathbf{Init}(t,\overline{L_2})}, \mathcal{H}o(L_2,t), \neg\mathcal{H}o(F,t), \mathcal{H}o(F',t) \mid t \in \mathbf{T} \text{ and } L_1 \cup L_2 \text{ is a supporting set of } F\}.$$

is equivalent to $D_2 =$

$$\{\mathbf{Causes}(t,l) \leftarrow \mathbf{Causes}(t,L_1), \mathcal{H}o(\overline{L_1},t), \overline{\mathbf{Causes}(t,\overline{L_2})}, \mathcal{H}o(L_2,t), \neg\mathcal{H}o(F,t), \mathcal{H}o(F',t) \mid t \in \mathbf{T} \text{ and } L_1 \cup L_2 \text{ is a supporting set of } F\}.$$

Proof:
The proof follows the same reasoning as that of theorem 7.3.1, with truth values determined by t instead of by St and A. □

Another result, adapted from theorem 7.3.2, is the following.

Theorem 7.3.5 *Given a set of direct and derived effect rules, a particular set of values for all fluents at a particular time* t, *and the set of actions occurring at* t, *the truth value of* $\mathbf{Init}(t,l)$ *is uniquely determined for all* l.

Proof:
The grounding of a specific set of effect rules is a definition in which (by its construction) the rules for $\mathbf{Init}(t,l)$ and $\mathbf{Causes}(t,l)$ only depend on the truth values $\mathcal{H}a(A,t)$ and $\mathcal{H}o(F,t)$. Given all fluents and actions at time t, these truth values are uniquely determined, hence the set of rules for all atoms of the form $\mathbf{Init}(t,l)$ or $\mathbf{Causes}(t,l)$ is unique. Moreover in this set of rules, $\mathbf{Init}(t,l)$ and $\mathbf{Causes}(t,l)$ only depend directly or indirectly on other atoms of the form $\mathbf{Init}(t,l')$ and $\mathbf{Causes}(t,l')$. Hence these rules form a complete definition on the atom domain $\{\mathbf{Init}(t,l), \mathbf{Causes}(t,l) \mid l \in \widehat{\mathbf{P}}\}$. $I_{\mathcal{D}_{init}}$ for this definition is unique, which proves the theorem. □

The following statement is an immediate corollary:

Corollary 7.3.2 *Given a completely determined initial state, a complete list of action occurrences and a particular set of effect rules, the truth value of all fluents at all time points is uniquely determined.*

Proof:
The proof follows from the above theorem by induction on the well-founded event topology, since the truth value of each fluent at the time of each event is uniquely determined by the truth values and initiations at the time of the preceding event. □

Finally, the results for acyclic rules and rules with only simple literals in the body also carry over immediately from the corresponding theorems above, with identical proofs. This illustrates how our semantics relates to existing approaches tackling the frame and ramification problems.

7.4 Examples of Various $\mathcal{ER}$-contributions

In this section, we formalise and discuss in more detail a few examples chosen to illustrate the claimed expressive power of $\mathcal{ER}$, in particular by illuminating some contributions we have not yet discussed in detail.

7.4.1 Simultaneous Actions

First of all, we have claimed that representing effects of simultaneous actions in $\mathcal{ER}$ can be done in a very natural and concise way. A nice example is found in [38]: a glass on a table spills its contents as soon as the table is in a non-horizontal position. This can be straightforwardly represented using complex derived effect rules, as

$$\textbf{initiating}\ \neg(up_l \leftrightarrow up_r)\ \textbf{causes}\ wet\ \textbf{if}\ on_table$$

where up_l, up_r represent that the left resp. right side of the table are lifted from the floor and wet that the table is wet. Recall that the grounding of this rule contains

$$\begin{array}{lcl}
\mathbf{Causes}(t, wet) & \leftarrow & \mathbf{Init}(t, up_l), \mathbf{Holds}(\neg up_r, t), \neg\mathbf{Init}(t, up_r), \\
& & \mathbf{Holds}(up_l \leftrightarrow up_r, t), \mathbf{Holds}(on_table, t). \\
\mathbf{Causes}(t, wet) & \leftarrow & \mathbf{Init}(t, up_r), \mathbf{Holds}(\neg up_l, t), \neg\mathbf{Init}(t, up_l), \\
& & \mathbf{Holds}(up_l \leftrightarrow up_r, t), \mathbf{Holds}(on_table, t). \\
\mathbf{Causes}(t, wet) & \leftarrow & \mathbf{Init}(t, \neg up_r), \mathbf{Holds}(up_l, t), \neg\mathbf{Init}(t, \neg up_l), \\
& & \mathbf{Holds}(up_l \leftrightarrow up_r, t), \mathbf{Holds}(on_table, t). \\
\mathbf{Causes}(t, wet) & \leftarrow & \mathbf{Init}(t, \neg up_l), \mathbf{Holds}(up_r, t), \neg\mathbf{Init}(t, \neg up_r), \\
& & \mathbf{Holds}(up_l \leftrightarrow up_r, t), \mathbf{Holds}(on_table, t). \\
\mathbf{Causes}(t, wet) & \leftarrow & \mathbf{Init}(t, up_l), \mathbf{Init}(t, \neg up_r), \\
& & \mathbf{Holds}(up_l \leftrightarrow up_r, t), \mathbf{Holds}(on_table, t). \\
\mathbf{Causes}(t, wet) & \leftarrow & \mathbf{Init}(t, up_r), \mathbf{Init}(t, \neg up_l), \\
& & \mathbf{Holds}(up_l \leftrightarrow up_r, t), \mathbf{Holds}(on_table, t).
\end{array}$$

for each t, where the last two rules can never have a true body since **Init** denotes strong initiation. Consider then two actions *lift_l* and *lift_r*:

$$\begin{array}{l}
lift_l\ \textbf{causes}\ up_l\ \textbf{if}\ true \\
lift_r\ \textbf{causes}\ up_r\ \textbf{if}\ true
\end{array}$$

Assuming the table is initially on the floor, executing either one of the actions will cause the water to spill, but if they are executed at the same time there is no spilling. It can be checked easily by evaluating the bodies of all effect rules that the given inductive definition leads to this intended conclusion in all cases. In addition, the representation by a complex derived effect rule is very concise and close to the natural language formulation of the effect. It also has the advantage that it avoids the explicit use of absences of initiations in the language: these only occur at the level of primitive rules, in certain sensible combinations. Finally, the derived effect rule is entirely independent of the actions which can influence the fluents involved. This modularity is of course required, in particular if there can be multiple actions lifting a side of the table. On the basis of these observations, which can be extended to applications with simultaneous actions in general, we argue that complex derived effect rules provide a correct and concise natural way to represent effects of simultaneous actions.

7.4.2 Incomplete Narrative Information

Another issue in $\mathcal{ER}$ which is orthogonal to the ramification issue we have discussed in much detail up to now, is the flexible representation of both complete and incomplete knowledge on the occurrence and order of actions and on the initial situation. To deal with this issue a simple first order logic theory is most appropriate: in general a FOL theory represents incomplete knowledge on all predicates. Hence, in an $\mathcal{ER}$-theory knowledge on all parts of the theory except on the effects of actions (which are represented by an inductive definition) is usually incomplete. However it is possible to explicitly state that knowledge about any part of the scenario is complete by using explicit Clark completion style axioms.[25] This approach offers maximal flexibility : it allows one to specify complete knowledge about very precise parts of the scenario (for example, about all occurrences of a certain action type, or about all action occurrences in a particular time interval), while leaving other parts partially specified.

As an example consider the well-known stolen car problem, formalised as follows:

$$\mathit{park}\ \textbf{causes}\ \mathit{parked}\ \textbf{if}\ \mathit{true}$$
$$\mathit{steal}\ \textbf{causes}\ \neg\mathit{parked}\ \textbf{if}\ \mathit{true}$$
$$\forall T : \mathbf{Happens}(\mathit{steal}, T) \rightarrow \mathbf{Holds}(\mathit{parked}, T)$$
$$\mathbf{Happens}(\mathit{park}, t_1) \wedge t_1 < t_2 \wedge \neg\mathbf{Holds}(\mathit{parked}, t_2)$$

[25] Extensions of $\mathcal{ER}$, in particular for dealing with delayed ramifications, will incorporate stronger principles than explicit completion; in particular an inductive definition semantics will be applied to the then arising theory of actions. We elaborate on this in section 7.9.

This specification represents incomplete information on action occurrences, and entails $\exists T : t_1 < T < t_2 \wedge \mathbf{Happens}(steal, T)$. We can assert complete knowledge on actions, i.e. that *park* is the only action, by adding

$$\forall A, T : \mathbf{Happens}(A, T) \leftrightarrow A = park \wedge T = t_1$$

In that case the specification is inconsistent.

Intuitively, one expects the above statements to be true. However, let us work out this example to clarify the details of the formal semantics. First of all, the grounding of the effect rules is

$$\begin{array}{l}\{\mathbf{Causes}(t, parked) \leftarrow \mathcal{H}a(park, t) \mid t \in \mathbf{T}\} \cup \\ \{\mathbf{Causes}(t, \neg parked) \leftarrow \mathcal{H}a(steal, t) \mid t \in \mathbf{T}\} \cup \\ \{\mathbf{Init}(t, parked) \leftarrow \mathbf{Causes}(t, parked), \neg \mathcal{H}o(parked, t) \mid t \in \mathbf{T}\} \cup \\ \{\mathbf{Init}(t, \neg parked) \leftarrow \mathbf{Causes}(t, \neg parked), \mathcal{H}o(parked, t) \mid t \in \mathbf{T}\},\end{array}$$

i.e. at each time point the definition of **Causes** consists of just two rules. The rules depend only on the truth values $\mathcal{H}a(park, t)$ and $\mathcal{H}a(steal, t)$, and these truth values determine the value of both $\mathbf{Causes}(t, parked)$ and $\mathbf{Causes}(t, \neg parked)$. We find that $\mathbf{Happens}(park, t_1)$ is true, hence so is $\mathbf{Causes}(t_1, parked)$: *parked* is weakly initiated. It then follows from the inertia axiom that

$$(\neg \exists T : t_1 \preceq T \prec t_2 \wedge (Causes(T, \neg parked)) \rightarrow \mathcal{H}o(parked, t_2).$$

Since it is given that $\neg \mathcal{H}o(parked, t_2)$, we find that

$$\exists T : t_1 \preceq T \prec t_2 \wedge Causes(T, \neg parked).$$

Then, $Causes(T, \neg parked) \leftrightarrow I_{\mathcal{D}_{init}}(\mathbf{Causes}(T, \neg parked))$ by the second condition in definition 7.3.14, so it follows that

$$\exists T : t_1 \preceq T \prec t_2 \wedge I_{\mathcal{D}_{init}}(\mathbf{Causes}(T, \neg parked)).$$

Since $I_{\mathcal{D}_{init}}(\mathbf{Causes}(t, \neg parked))$ is true if and only if $\mathcal{H}a(steal, t)$ is true, it follows that

$$\exists T : t_1 < T < t_2 \wedge \mathbf{Happens}(steal, T)$$

which is what we intended to prove. Given this result, it is also obvious that adding $\forall A, T : \mathbf{Happens}(A, T) \leftrightarrow A = park \wedge T = t_1$ to the theory leads to inconsistency.

7.5 Mapping $\mathcal{ER}$ to OLP Event Calculus

We now map $\mathcal{ER}$ theories to a variant of the OLP Event Calculus, showing which sublanguage of the Event Calculus is retained in $\mathcal{ER}$ and how the constructs in both languages are related to each other.

The mapping is not very complicated, as $\mathcal{ER}$ and OLP Event Calculus are based on the same principles (inductive definitions, first order logic, and events in linear time). As semantics for open logic programs we adopt the justification semantics described in Chapter 2.

The variant of the Event Calculus we use differs in the following respects from the one we have used up to now:

- The predicates *initially* and *holds* are defined for complex fluent formulae rather than only for simple fluents.

- The predicate *causes*, which takes fluent literals as second argument, replaces the predicates *initiates* and *terminates*: $causes(t, f)$ replaces $initiates(t, f)$ and $causes(t, \neg f)$ replaces $terminates(t, f)$.

- The distinction between events and the time of their occurrence, which we have used throughout this thesis though it was optional, is dropped. Hence, the formulae $happens(e, t)$ and $act(e, a)$ can be contracted into one formula $happens(a, t)$. Also, *causes* then takes a time point as its first argument rather than an event.

- The initial state is represented differently: it is now assumed that this is the state of the world before any events. The construction of a *start* event which initiates all initially true fluents, which we have used up to now for simplicity reasons, is dropped entirely and replaced by a cleaner treatment of *initially*. This leads to some small (and evident) modifications to the frame axiom.

It should be obvious that these are mostly syntactic modifications introduced to simplify the mapping below. They have no impact on the essence of the Event Calculus.

7.5.1 Mapping Predicates

First of all, we establish a relation between the predicates in the two formalisms. In the open logic program, we use the predicates *happens*, *holds*, $\leq$, *initially* and *causes*. The relation to the $\mathcal{ER}$ predicates is the following, given a an action constant, t, t' time constants, l a fluent literal and F a

fluent formula[26]:

$$\begin{array}{rcl} happens(a,t) & \leftrightarrow & \mathcal{H}a(a,t) \\ holds(F,t) & \leftrightarrow & \mathcal{H}o(F,t) \\ t \leq t' & \leftrightarrow & t \preceq t' \\ initially(F) & \leftrightarrow & \mathcal{I}nitially(F) \\ causes(t,l) & \leftrightarrow & Causes(t,l) \end{array}$$

In the open logic program, the predicates *causes*, *holds*, and the *initially* predicate for complex fluent formulae are defined predicates, while the other predicates are open. We deal with the complex *holds* and *initially* atoms right away, so that in the sequel we only need to handle *holds* and *initially* atoms containing fluent atoms. Defining the complex atoms can easily be done inductively, by the following clauses:

$$\begin{array}{lcl} holds(F_1 \wedge F_2, T) & \leftarrow & holds(F_1, T), holds(F_2, T). \\ holds(F_1 \vee F_2, T) & \leftarrow & holds(F_1, T). \\ holds(F_1 \vee F_2, T) & \leftarrow & holds(F_2, T). \\ holds(\neg F_1, T) & \leftarrow & \neg holds(F_1, T). \\ holds(true, T). & & \end{array}$$

$$\begin{array}{lcl} initially(F_1 \wedge F_2) & \leftarrow & initially(F_1), initially(F_2). \\ initially(F_1 \vee F_2) & \leftarrow & initially(F_1). \\ initially(F_1 \vee F_2) & \leftarrow & initially(F_2). \\ initially(\neg F_1) & \leftarrow & \neg initially(F_1). \\ initially(true). & & \end{array}$$

For simple *holds* atoms a definition is given below, while *initially* for fluent atoms is an open predicate. Observe that we need to add explicit clauses for the fluent formula *true*. This corresponds to the condition on $\mathcal{ER}$-interpretations that $\mathcal{H}o(true, t)$ and $\mathcal{I}nitially(true)$ must be true.

7.5.2 Dealing with the Domain-Independent Conditions

Next, let us have a look at the general conditions on $\mathcal{ER}$ interpretations and how these relate to OLP Event Calculus. Concerning the time structure, we can impose like in $\mathcal{ER}$ that time is isomorphic to the real numbers. In OLP Event Calculus, we usually impose the weaker condition that time

[26] Note that the Init predicate has no Event Calculus counterpart, but is redundant since $\mathcal{I}nit(t,l) \leftrightarrow Causes(t,l) \wedge \mathcal{H}o(\bar{l},t)$

points are linearly ordered, by the FOL axioms

$$\begin{array}{c}\forall T_1, T_2 : (T_1 \leq T_2 \wedge T_2 \leq T_1) \rightarrow T_1 = T_2 \\ \forall T_1, T_2, T_3 : (T_1 \leq T_2 \wedge T_2 \leq T_3) \rightarrow T_1 \leq T_3 \\ \forall T_1, T_2 : T_1 \leq T_2 \vee T_2 \leq T_1.\end{array}$$

Evidently the real numbers satisfy this condition. We can assume time to be isomorphic to them.

The well-founded event topology can be imposed as follows. We define the predicate *next* as

$$\begin{array}{lcl} next(T_1, T_2) & \leftarrow & happens(A_1, T_1), happens(A_2, T_2), T_1 < T_2, \\ & & \neg int_events(T_1, T_2). \\ int_events(T_1, T_2) & \leftarrow & happens(A_3, T_3), T_1 < T_3, T_3 < T_2. \end{array}$$

and its transitive closure *before* as

$$\begin{array}{lcl} before(T_1, T_2) & \leftarrow & next(T_1, T_2). \\ before(T_1, T_2) & \leftarrow & next(T_1, T_3), before(T_3, T_2). \end{array}$$

The axiom

$$[(T_1 < T_2) \wedge happens(A_1, T_1) \wedge happens(A_2, T_2)] \leftrightarrow before(T_1, T_2)$$

then imposes that there are only a finite number of events between each two events. In addition, the axiom

$$\exists E_{start} : \forall A, T : [happens(A, T) \rightarrow \neg(T < E_{start})]$$

ensures that there is a first event. Hence there is an order-preserving isomorphism between events and a subset of the natural numbers. This is the desired topology on events. In addition, we need to embed this topology in the time structure such that in case of an infinite number of events, there are no time points after the entire infinite sequence. This is imposed by the axiom

$$\begin{array}{c}\forall T : ([\exists A, E : (happens(A, E) \wedge E < T \wedge \neg int_events(E, T))] \\ \vee[\neg exists A, E : (happens(A, E) \wedge E < T)])\end{array}$$

Given that time is isomorphic to the real numbers, the well-founded event topology condition defined earlier is imposed by the above theory.

The initiation consistency condition is represented by the Event Calculus FOL axiom

$$\forall T, F : \neg causes(T, F) \vee \neg causes(T, \neg F)$$

which is equivalent to $\forall T, F : \neg Causes(T, F) \vee \neg Causes(T, \neg F)$. Then we still need to formalise the definition of the initial state and the inertia axiom. This is achieved by providing the following definition for *holds*:

$$
\begin{array}{lcl}
holds(F, T) & \leftarrow & T \leq e_{start}, initially(F). \\
holds(F, T) & \leftarrow & causes(T', F), T' < T, \neg clipped(T', F, T). \\
clipped(T', F, T) & \leftarrow & causes(T'', \neg F), T' \leq T'', T'' < T. \\
T' < T & \leftarrow & T' \leq T, T' \neq T.
\end{array}
$$

The first clause for *holds* defines the initial state, the other clause is the Event Calculus frame axiom, which will further on be proven equivalent to the inertia axiom in $\mathcal{ER}$ given the other assumptions. Note that the event e_{start} is the first actual event, and not the artificial *start* event used in OLP Event Calculus earlier in this thesis. There is no such *start* event in the Event Calculus variant we use here, as time is considered unbounded in the past (one can say the *start* event is infinitely far in the past).

7.5.3 Mapping Π_p to FOL Axioms

The Π_p part of an $\mathcal{ER}$ theory contains general FOL formulae constructed from **Holds**, **Happens**, $\leq$ and **Initially** atoms using connectives and quantifiers. We can map these formulae to FOL axioms straightforwardly by replacing $holds(F, \tau)$ for $\mathbf{Holds}(F, \tau)$, $happens(\alpha, \tau)$ for $\mathbf{Happens}(\alpha, \tau)$, $\tau_1 \leq \tau_2$ for itself and $initially(F)$ for $\mathbf{Initially}(F)$. Given the mapping of predicates above it is easy to check that this mapping preserves the semantics.

7.5.4 Mapping Π_e to Program Clauses

As a guideline for mapping the effect rules to OLP, we can look at the definition of grounding in the semantics of $\mathcal{ER}$: as we had to do in the grounding, we need to deal with complex initiations, and this can be done in the same way:

A direct effect rule a **causes** l **if** F is mapped to the clause

$$causes(T, l) \leftarrow happens(a, T), holds(F, T).$$

A derived effect rule **initiating** F **causes** l **if** F' is mapped to the set of clauses

$$
\begin{array}{lcl}
\{causes(T, l) & \leftarrow & \bigwedge_{l \in L_1} causes(T, l), \\
 & & \bigwedge_{l' \in L_2} (holds(l, T), \neg causes(T, \overline{l'})) \\
 & & \neg holds(F, T), holds(F', T). \\
 & | & L_1 \cup L_2 \text{ is a supporting set of } F\}
\end{array}
$$

Both results of the mapping correspond to the respective groundings of the rules, but compacted to a finite number of clauses by using universal quantification over time points instead of different clauses for each t. A minor difference is that the predicates *happens* and *holds* occur explicitly in the clauses instead of their truth values. A more noteworthy difference is that in the body of the OLP clauses, we use the *causes* predicate which denotes weak initiation, whereas in $\mathcal{ER}$ **Init**, i.e. strong initiation, is used in the inductive definition rules. However we have proven the equivalence of the two corresponding ways of defining *Causes* in $\mathcal{ER}$ before.

Note that there is no predicate denoting strong initiation in the OLP theory. Therefore the $\mathcal{ER}$-rules defining **Init** in terms of **Causes** have no equivalent OLP clauses. They have been compiled into the clauses for *causes*.

Finally, observe that we can consider all fluents and time points in the domain to be known, so that we only need to consider Herbrand interpretations. Hence we can use the definition of proof tree for Herbrand interpretations in the justification semantics, which shows a more immediate correspondence with the inductive definition concept of proof tree.

7.5.5 A Detailed Example and some Remarks

Taking everything together, we now show the mapping of the stolen car problem presented earlier:

$$\begin{gathered} \textit{park} \ \textbf{causes} \ \textit{parked} \ \textbf{if} \ \textit{true} \\ \textit{steal} \ \textbf{causes} \ \neg\textit{parked} \ \textbf{if} \ \textit{true} \\ \forall T : \textbf{Happens}(\textit{steal}, T) \rightarrow \textbf{Holds}(\textit{parked}, T) \\ \textbf{Happens}(\textit{park}, t_1) \wedge t_1 < t_2 \wedge \neg\textbf{Holds}(\textit{parked}, t_2) \end{gathered}$$

First of all, we have the general formulae, i.e. the Event Calculus frame axiom

$$\begin{array}{lcl} \textit{holds}(F,T) & \leftarrow & T < e_{start}, \textit{initially}(F). \\ \textit{holds}(F,T) & \leftarrow & \textit{causes}(T',F), T' < T, \neg\textit{clipped}(T',F,T). \\ \textit{clipped}(T',F,T) & \leftarrow & \textit{causes}(T'',\neg F), T' \leq T'', T'' < T. \\ T' < T & \leftarrow & T' \leq T, T' \neq T. \end{array}$$

and the constraints on time

$$\begin{gathered} \forall T_1, T_2 : (T_1 \leq T_2 \wedge T_2 \leq T_1) \rightarrow T_1 = T_2 \\ \forall T_1, T_2, T_3 : (T_1 \leq T_2 \wedge T_2 \leq T_3) \rightarrow T_1 \leq T_3 \\ \forall T_1, T_2 : T_1 \leq T_2 \vee T_2 \leq T_1 \end{gathered}$$

The effect rules are mapped to clauses

$$\begin{array}{lcl} causes(T, parked) & \leftarrow & happens(park, T). \\ causes(T, \neg parked) & \leftarrow & happens(steal, T). \end{array}$$

The other formulae are simply mapped to FOL formulae

$$\forall T : (happens(steal, T) \rightarrow holds(parked, T))$$
$$happens(park, t_1) \wedge t_1 < t_2 \wedge \neg holds(parked, t_2)$$

The predicates $<$, *initially* and *happens* are undefined, and we ignore complex *holds* and *initially* atoms as they do not occur in this example. The formula which was to be entailed is mapped to $\exists T : t_1 < T < t_2 \wedge happens(steal, T)$. Of course it is easy to check that this formula is entailed, following the same reasoning as in the $\mathcal{ER}$ formalisation.

In $\mathcal{ER}$ we also added an explicit completion on action occurrences to this formalisation, which led to inconsistency. We can do the same in the OLP formalisation, but note that another and simpler option is to make *happens* a defined predicate, defined by the fact $happens(park, t_1)$. In general, if there is complete knowledge on *happens* or *initially* in any scenario, we can choose to make these predicates defined instead of adding explicit completions. This is only important for conciseness of representation and for reasoning performance: semantically there is no difference.

7.5.6 Equivalence Proof

To prove equivalence of the two theories, we proceed as follows. First, observe that as indicated above, the initiation consistency, linear order and well-founded event topology conditions are properly axiomatised. Now, take any interpretation of the undefined predicates (*initially* for fluent atoms, *happens* and $\leq$), and the same interpretation for $\mathcal{I}nitially$ for fluent atoms, $\mathcal{H}a$ and $\preceq$. From these we can compute the truth values of all other atomic formulae using on the one hand the program clauses in OLP, and on the other hand the inductive definitions and the inertia axiom in $\mathcal{ER}$. We will prove that this yields the same results in both formalisms, which leaves us with the same models of Π_e of each theory. Finally, the FOL axioms and the corresponding theory Π_p and general axioms of $\mathcal{ER}$ will be checked in all of those Π_e-models, retaining only those interpretations that satisfy all of the axioms. Given the trivial correspondence between Π_p formulae and general $\mathcal{ER}$ axioms with FOL axioms in OLP, it is obvious that the same interpretations are retained in both formalisms by this step.

So what we still need to prove is that given *initially* for fluent atoms, *happens* and $\leq$, and their exact counterparts in $\mathcal{ER}$, we find the same

truth value for all other atoms in both formalisms. For complex *initially* and $\mathcal{I}nitially$ atoms this is trivial since we use the same inductive definition in terms of simple atoms in both formalisms. This leaves us with simple and complex *holds* atoms and *causes* atoms.

The clauses of *holds* and *causes* are mutually recursive. $holds(f,t)$ depends on the truth values of $causes(f,t')$ and $causes(\neg f,t')$ for $t' < t$. $holds(F,t)$ depends on atoms $holds(f,t)$. $causes(l,t)$ may depend on $holds(F,t)$ (and $causes(l',t)$) for any F and l'. Given a well-founded event topology, we can prove equivalence of $holds(f,t)$ with $\mathcal{H}o(f,t)$, of $holds(F,t)$ with $\mathcal{H}o(F,t)$ and of $causes(t,l)$ with $Causes(t,l)$ by induction on events: first we prove that $holds(F,t)$ coincides with $\mathcal{H}o(F,t)$ for all time points before the first event, then for each event e we prove consecutively that $causes(e,l) \leftrightarrow Causes(e,l)$, that for all time points t between e and the next event (including it), $holds(f,t) \leftrightarrow \mathcal{H}o(f,t)$, and that for all these time points $holds(F,t) \leftrightarrow \mathcal{H}o(F,t)$. By induction on the well-founded event topology it then follows that $holds(F,t) \leftrightarrow \mathcal{H}o(F,t)$ for all time points and that $causes(t,l) \leftrightarrow Causes(t,l)$ for all events (at non-event time points it is easy to see that $causes(t,l)$ as well as $Causes(t,l)$ are false from their definitions: no proof tree exists without *happens* atoms in one or more leaves).

We now proceed with the inductive proof.

1. Induction base:

   For all F and for all $t \leq e_{start}$, in $\mathcal{ER}$ it holds that $\mathcal{I}nitially(F) \leftrightarrow \mathcal{H}o(F,t)$. Likewise in OLP we know from the definition of *holds* that if f is a fluent atom and $t \leq e_{start}$, then $holds(f,t) \leftrightarrow initially(f)$ since the second clause can never have a true body and the first clause simplifies to $holds(f,t) \leftarrow initially(f)$. Hence, for all fluent atoms f, we know that $holds(f,t) \leftarrow \mathcal{H}o(f,t)$ for $t \leq e_{start}$. The definition of complex *holds* atoms in terms of simple ones is the same in OLP and $\mathcal{ER}$, so the result generalises to $holds(F,t) \leftarrow \mathcal{H}o(F,t)$ for all fluent formulae F and all $t \leq e_{start}$.

2. Assume $holds(F,t) \leftrightarrow \mathcal{H}o(F,t)$ for all F and a particular t. The definition of $causes(t,l)$ for a certain t depends only on $happens(a,t)$ and $holds(F,t)$ and on itself ($causes(t,l')$), just like the definition of $\mathbf{Causes}(t,l)$ in $\mathcal{ER}$ depends only on truth values $\mathcal{H}a(a,t)$ and $\mathcal{H}o(F,t)$. Moreover, the definitions are the same, as we have shown above. Hence we find that $causes(t,l) \leftrightarrow I_{\mathcal{D}_{init}}(\mathbf{Causes}(t,l))$.

3. Assume $causes(t,l) \leftrightarrow Causes(t,l)$ for a particular event t. We need to prove that for all t' after t and before or equal to the next event

t'', $holds(f,t') \leftrightarrow \mathcal{H}o(f,t')$. Take an arbitrary f and t'. The inertia axiom in $\mathcal{ER}$ yields for f the formula

$$t \prec t' \wedge \neg\exists t_3 : (t \preceq t_3 \prec t' \wedge Causes(t_3, \neg f))$$
$$\rightarrow \mathcal{H}o(f,t') \leftrightarrow Causes(t,f) \vee \mathcal{H}o(f,t)$$

which can be simplified to

$$\neg Causes(t,\neg f) \rightarrow (\mathcal{H}o(f,t') \leftrightarrow Causes(t,f) \vee \mathcal{H}o(f,t))$$

and then to

$$[\mathcal{H}o(f,t') \wedge \neg Causes(t,\neg f)]$$
$$\leftrightarrow [(Causes(t,f) \vee \mathcal{H}o(f,t)) \wedge \neg Causes(t,\neg f)]$$

It follows that

$$\mathcal{H}o(f,t') \leftarrow$$
$$([Causes(t,f) \wedge \neg Causes(t,\neg f)] \vee [\mathcal{H}o(f,t) \wedge \neg Causes(t,\neg f)])$$

Similarly, we find for $\neg f$ that

$$\mathcal{H}o(\neg f,t') \leftarrow$$
$$([Causes(t,\neg f) \wedge \neg Causes(t,f)] \vee [\mathcal{H}o(\neg f,t) \wedge \neg Causes(t,f)])$$

hence because $\mathcal{H}o(\neg f,t) \leftrightarrow \neg\mathcal{H}o(f,t)$, that

$$\neg\mathcal{H}o(f,t') \leftarrow$$
$$([Causes(t,\neg f) \wedge \neg Causes(t,f)] \vee [\neg\mathcal{H}o(f,t) \wedge \neg Causes(t,f)])$$

Contraposition of this formula yields

$$\mathcal{H}o(f,t') \rightarrow ([\neg Causes(t,\neg f) \vee Causes(t,f)] \wedge [\mathcal{H}o(f,t) \vee Causes(t,f)])$$

which can be simplified to

$$\mathcal{H}o(f,t') \rightarrow ([\neg Causes(t,\neg f) \wedge \mathcal{H}o(f,t)] \vee Causes(t,f))$$

Together with the formula obtained in the previous paragraph, which can be simplified to

$$\mathcal{H}o(f,t') \leftarrow ([\neg Causes(t,\neg f) \wedge \mathcal{H}o(f,t)] \vee Causes(t,f))$$

we obtain the equivalence

$$\mathcal{H}o(f,t') \leftrightarrow (Causes(t,f) \vee [\mathcal{H}o(f,t) \wedge \neg Causes(t,\neg f)])$$

which is equivalent to (given the assumption of the inductive step)

$$\mathcal{H}o(f,t') \leftrightarrow ([causes(t,f) \vee [holds(f,t) \wedge (\neg causes(t,\neg f)]$$

On the other hand, the definition of *holds* entails

$$holds(f,t') \leftrightarrow \exists t^* : [causes(t^*,f) \wedge t^* < t' \wedge \neg\exists t'' : (causes(t'',\neg f) \wedge t^* \leq t'' \wedge t'' < t')]$$

since completion semantics is strictly weaker than justification semantics. A possible t^* in that formula can either be before t or equal to t, taking into account the fact that $t < t'$, that there is no event between t and t' and that time is a linear order. In the first case, the condition

$$causes(t^*,f) \wedge t^* < t' \wedge \neg\exists t'' : (causes(t'',\neg f) \wedge t^* \leq t'' \wedge t'' < t')$$

can be written as

$$causes(t^*,f) \wedge t^* < t' \wedge t' < t \wedge \neg\exists t'' : (causes(t'',\neg f) \wedge t^* \leq t'' \wedge t'' < t) \wedge \neg causes(t,\neg f)$$

which is equivalent to $holds(f,t) \wedge \neg causes(t,\neg f)$; in the second case that same condition simply reads $causes(t,f)$. We obtain

$$holds(f,t') \leftrightarrow [holds(f,t) \wedge \neg causes(t,\neg f)] \vee causes(t,f)$$

and this formula combined with the result for $\mathcal{H}o$ yields

$$holds(f,t) \leftrightarrow \mathcal{H}o(f,t)$$

4. The definition of complex *holds* atoms in terms of simple ones is the same in OLP and $\mathcal{ER}$, so if $holds(f,t) \leftrightarrow \mathcal{H}o(f,t)$ for all fluent atoms f and the above defined time points t, then $holds(F,t) \leftarrow \mathcal{H}o(F,t)$ for all fluent formulae F and all those t.

This completes the proof by induction, and thereby the entire equivalence proof of the OLP Event Calculus theory and the $\mathcal{ER}$ theory.

7.6 Dealing with Nondeterminism in $\mathcal{ER}$

We now turn our attention to the representation of nondeterministic effects of actions. We assume in this discussion that the outcome of a nondeterministic action is one of a number of possible effects (possibly the empty

effect), that the set of possible effects can be dependent on the state of the world, and that the actual effect can be different for different instances of the action.

As an example, consider a variant of the Yale Shooting Problem in which firing a loaded gun may nondeterministically hit a turkey in the head (killing it) or in the wing (breaking that wing). A representation of this effect could be a rule

$$shoot \textbf{ causes } \neg alive \vee broken_wing \textbf{ if } loaded$$

which is a generalisation of a direct effect rule with a disjunction in the head. In general, we choose the following syntax for nondeterministic effect rules:

$$a \textbf{ causes } D \textbf{ if } F$$

where a is an action, F a fluent formula, and D a disjunction of fluent literals.[27] For reasons that will be explained below we will use the symbol $|$ instead of $\vee$ to denote disjunction in D.

The semantics of such a nondeterministic effect rule is that when the action a is executed while F holds, one of the disjuncts in D is (weakly) initiated. Let us try to define this more precisely for the above example: we can say that when a loaded gun is fired, *broken_wing* or $\neg alive$ should be initiated. But should this "or" be inclusive or exclusive? We will show that neither option is satisfactory.

Clearly an inclusive or is unintended: the bullet should hit the turkey only in one place, not both. This seems to leave the exclusive or as only plausible reading. However, the following examples show that this reading is also unintended. First, consider the ramification that a turkey dies when its wing gets broken while it is flying[28], as represented by the derived effect rule

$$\textbf{initiating } broken_wing \textbf{ causes } \neg alive \textbf{ if } flying$$

The intuitive result of shooting a flying turkey is that it always dies, either by the shot in the head or as a result of its broken wing. However, reading "or" in the nondeterministic effect rule as exclusive, we would reach the unintended conclusion that its wing can never get broken since it already dies.

A similar problem arises in the presence of simultaneous actions: assume that apart from the original hunter, there is now also an expert hunter who always kills the turkey when shooting. Then assume both hunters shoot at

[27] Below we further generalise the formulae allowed as D.

[28] We ignore for now other evident relations between *flying*, *alive* and *broken_wing* as they are not relevant in this discussion.

the same time. We expect the nondeterministic shot to either break the turkey's wing or to kill it, while the expert shot definitely kills the turkey. Yet reading "or" as exclusive leads to the conclusion that, since *alive* is already terminated by the expert shot, the nondeterministic shot cannot break the turkey's wing at the same time.

How should we read nondeterministic rules then? In the above examples, the intuitive reading is that the *shoot* action has exactly one effect, either killing the turkey or breaking its wing. However, other sources (either ramifications or effects of other simultaneous actions) may cause the second effect as well, so that it is not guaranteed that there is only one effect, even if only one may be caused by the given direct effect rule. In general, recall that we view effect rules as describing the propagation of effects due to some physical or logical force. The idea underlying nondeterministic actions is that this force may act in one of several possible ways. This boils down to saying that a rule

$$a \textbf{ causes } f_1 \mid f_2 \textbf{ if } F$$

is equivalent to

$$a \textbf{ causes } f_1 \textbf{ if } F \oplus a \textbf{ causes } f_2 \textbf{ if } F$$

or in other words a nondeterministic effect rule has at all times exactly the same effect as one rule obtained from it by retaining only one of its disjuncts. The thus obtained rules will be called the disjunctive components of the nondeterministic effect rules. Generalising this, we say an interpretation satisfies a definition $\mathcal{D}$ at a particular time point iff it satisfies *any* definition obtained from $\mathcal{D}$ by replacing each nondeterministic effect rule by one of its disjunctive components.

As an example, consider the rules formalising the effect of shooting a flying turkey:

shoot **causes** ¬*alive* | *broken_wing* **if** *loaded*
initiating *broken_wing* **causes** ¬*alive* **if** *flying*

This nondeterministic definition is satisfied if at each time point at least one of the deterministic definitions

shoot **causes** ¬*alive* **if** *loaded*
initiating *broken_wing* **causes** ¬*alive* **if** *flying*

or

shoot **causes** *broken_wing* **if** *loaded*
initiating *broken_wing* **causes** ¬*alive* **if** *flying*

is satisfied. In the first case, we get the effect that the turkey is killed by the shot, in the second case that its wing gets broken and it dies as a result.[29]

Before formalising the above intuitive semantics, we want to extend the syntax of nondeterministic effect rules. In particular, we want to allow for a nondeterministic choice of sets of effects, rather than of single effects. This is achieved by defining nondeterministic effect rules as formulae

$$a \textbf{ causes } D \textbf{ if } F$$

where a is an action, F a fluent formula, and D a disjunction of conjunctions of fluent literals. Note that "true" can be used to denote the empty effect, since true is always weakly initiated.

One may wonder why we do not allow D to be any propositional fluent formula F', since any F' can be written in disjunctive normal form. The reason is that equivalent FOL formulae may lead to non-equivalent effect rules: for example p is equivalent to $(p \wedge q) \vee (p \wedge \neg q)$, but there is an important difference between the rules a **causes** p **if** *true* and a **causes** $(p \wedge q) \mid (p \wedge \neg q)$ **if** *true*: according to the first rule a initiates p and leaves q alone, while according to the second rule a initiates p and can nondeterministically initiate or terminate q. This is the reason why $\mid$ should not be read as classical disjunction, and why we should not rely on reducing general propositional formulae to some normal form.

We are now ready to define the semantics of nondeterministic effect rules. To this end we extend the notion of grounding as follows : a grounding of a nondeterministic effect rule is obtained from the groundings of its disjunctive components by collecting, for each time point t, from the grounding of one arbitrary disjunctive component, all the rules containing t. Formally:

Definition 7.6.1 (nondeterministic grounding (direct))
The restriction G^t of a set of primitive definition rules G to a time point t is the set of all rules of G in which t occurs.

The grounding of a conjunctive direct effect rule

$$a \textbf{ causes } \bigwedge_{i=1\ldots n} l_i \textbf{ if } F$$

is the set $\bigcup_{i=1\ldots n} G_i$, where each G_i is the grounding of the direct effect rule a **causes** l_i **if** F.

[29] It is essential that at different time points, different deterministic definitions can be satisfied: the nondeterministic action can have a different outcome each time. It can also occur by coincidence that both definitions are satisfied at a particular moment (if their effects happen to be the same). This is for example trivially the case when the gun is not loaded.

A grounding of a nondeterministic direct effect rule

$$a \textbf{ causes } C_1 \mid \ldots \mid C_m \textbf{ if } F$$

is any set $\bigcup_{t \in \mathbf{T}} G^t_{j_t}$*, where* $1 \leq j_t \leq m$ *for each* $t \in \mathbf{T}$*,* G_{j_t} *is the grounding of the conjunctive effect rule* a **causes** C_{j_t} **if** F*, and* $G^t_{j_t}$ *is the restriction of* G_{j_t} *to* t*.*

A grounding of $\Pi_e = \{r_k \mid 1 \leq k \leq l\}$ *is any definition* $\mathcal{D}_{init} = \mathcal{D}_g \cup \{\textbf{Init}(t,l) \leftarrow \textbf{Causes}(t,l), \neg\mathcal{H}o(l,t) \mid t \in \mathbf{T}, l \in \widehat{\mathbf{P}}\}$*, where* $\mathcal{D}_g$ *is any set* $\bigcup_{k=1\ldots l} G_k$ *with each* G_k *a grounding of* r_k*.*

$I \models \Pi_e$ *iff for any grounding* $\mathcal{D}_{init}$ *of* Π_e*,* $\forall t \in \mathbf{T}, l \in \widehat{\mathbf{F}}$ *:*
$\mathcal{I}nit(t,l) \leftrightarrow I_{\mathcal{D}_{init}}(\textbf{Init}(t,l))$ *and* $\mathcal{C}auses(t,l) \leftrightarrow I_{\mathcal{D}_{init}}(\textbf{Causes}(t,l))$*.*

The groundings of the above example definition are sets containing for each time point $t \in \mathbf{T}$, either

$$\begin{array}{lcl}\textbf{Causes}(t, \neg alive) & \leftarrow & \mathcal{H}a(shoot,t), \mathcal{H}o(loaded,t).\\ \textbf{Causes}(t, \neg alive) & \leftarrow & \textbf{Init}(t, broken_wing), \mathcal{H}o(flying,t).\end{array}$$

or

$$\begin{array}{lcl}\textbf{Causes}(t, broken_wing) & \leftarrow & \mathcal{H}a(shoot,t), \mathcal{H}o(loaded,t).\\ \textbf{Causes}(t, \neg alive) & \leftarrow & \textbf{Init}(t, broken_wing), \mathcal{H}o(flying,t).\end{array}$$

The syntax of derived effect rules can be extended in the same way as that of direct effect rules, which leads to rules

$$\textbf{initiating } F \textbf{ causes } D \textbf{ if } F'$$

representing nondeterministic ramifications. The semantics of such rules is defined by extending the definition of grounding in the same way as for direct effect rules. Formally:

Definition 7.6.2 (nondeterministic grounding (derived))
The grounding of a conjunctive derived effect rule

$$\textbf{initiating } F \textbf{ causes } \bigwedge_{i=1\ldots n} l_{f_i} \textbf{ if } F'$$

is the set $\bigcup_{i=1\ldots n} G_i$*, where each* G_i *is the grounding of the derived effect rule* **initiating** F **causes** l_{f_i} **if** F'*.*

A grounding of a nondeterministic derived effect rule

$$\textbf{initiating } F \textbf{ causes } C_1 \mid \ldots \mid C_m \textbf{ if } F'$$

is any set $\bigcup_{t \in \mathbf{T}} G^t_{j_t}$*, where* $\forall t :\ 1 \leq j_t \leq m$*,* G_{j_t} *is the grounding of the conjunctive derived effect rule* **initiating** F **causes** C_{j_t} **if** F'*, and* $G^t_{j_t}$ *is the restriction of* G_{j_t} *to* t*.*

As an example, consider again the effect of shooting a flying turkey in the wing. Maybe it is unpredictable if the turkey will die as a result or not: it may for example depend on how high the turkey is flying. This can be represented by replacing the derived effect rule in the above example by

$$\textbf{initiating } \textit{broken_wing} \textbf{ causes } \neg \textit{alive} \mid \textit{true} \textbf{ if } \textit{flying}$$

where the disjunct *true* indicates the absence of any additional effect. The groundings of the resulting definition are now sets containing for each time point $t \in \mathbf{T}$, one of the four definitions

$$\begin{array}{lll}
\mathbf{Causes}(t, \neg alive) & \leftarrow & \mathcal{H}a(shoot, t), \mathcal{H}o(loaded, t). \\
\mathbf{Causes}(t, \neg alive) & \leftarrow & \mathbf{Init}(t, broken_wing), \mathcal{H}o(flying, t).
\end{array}$$

$$\begin{array}{lll}
\mathbf{Causes}(t, broken_wing) & \leftarrow & \mathcal{H}a(shoot, t), \mathcal{H}o(loaded, t). \\
\mathbf{Causes}(t, \neg alive) & \leftarrow & \mathbf{Init}(t, broken_wing), \mathcal{H}o(flying, t).
\end{array}$$

$$\begin{array}{lll}
\mathbf{Causes}(t, \neg alive) & \leftarrow & \mathcal{H}a(shoot, t), \mathcal{H}o(loaded, t). \\
\mathbf{Causes}(t, true) & \leftarrow & \mathbf{Init}(t, broken_wing), \mathcal{H}o(flying, t).
\end{array}$$

$$\begin{array}{lll}
\mathbf{Causes}(t, broken_wing) & \leftarrow & \mathcal{H}a(shoot, t), \mathcal{H}o(loaded, t). \\
\mathbf{Causes}(t, true) & \leftarrow & \mathbf{Init}(t, broken_wing), \mathcal{H}o(flying, t).
\end{array}$$

in which the rules for $\mathbf{Causes}(t, true)$ evidently have no effect.

The above definitions show that a syntax and semantics for nondeterministic ramifications can be obtained as a straightforward extension of our existing constructs. At first sight it may not be clear if nondeterministic ramifications are of great practical importance, but it should be noted that they occur at least implicitly in some approaches to the ramification problem, as we will discuss below.

7.6.1 Mapping Nondeterministic Effect Rules to OLP

Dealing with nondeterministic effect rules can essentially be done in the same way as dealing with deterministic rules, basing the mapping on the groundings of the rules. A complication is that OLP does not contain disjunctive rules. We could of course extend the syntax of OLP with disjunctive rules, in which case the mapping would be straightforward. But this extension of the OLP syntax is not necessary: an OLP theory with the same models as the nondeterministic $\mathcal{ER}$ theory can be cleanly constructed by introducing a number of auxiliary "degree of freedom" predicates. The technique has been discussed briefly in Chapter 5 in the context of nondeterministic actions in Event and Situation Calculus. We here show how

it applies to general effect rules, thereby also showing the precise semantics of theories containing such predicates (i.e. exactly the above defined semantics for nondeterministic effect rules).

As defined before, a nondeterministic rule has multiple groundings corresponding to its multiple possible outcomes. An interpretation is a model of a set of effect rules if at each time point it satisfies any set of definition rules containing exactly one grounding of each effect rule. The idea now is the following: for each effect rule, *all* of its groundings are converted to OLP like in section 7.5, but an additional literal is inserted in the body of each clause such that for any interpretation of the additional literals only one clause body is not trivially false. Moreover this one clause body then needs to be equivalent to the corresponding original clause (without the added literal). Which of the clause bodies is not trivially false, depends on the truth values of the additional literals. These literals should of course take on exactly those combinations of values needed to "select" all clauses once.

As an example, assume a simple nondeterministic action with two possible outcomes, described by a **causes** $f_1 \mid f_2$ **if** *true*. Its two disjunctive components would yield mappings

$$\begin{array}{rcl} causes(T, f_1) & \leftarrow & happens(a, T). \\ causes(T, f_2) & \leftarrow & happens(a, T). \end{array}$$

We add literals to the rules such that only one rule body can be true in one interpretation:

$$\begin{array}{rcl} causes(T, f_1) & \leftarrow & happens(a, T), choice(c_r, 1, T). \\ causes(T, f_2) & \leftarrow & happens(a, T), choice(c_r, 2, T). \end{array}$$

with FOL axioms

$$\forall T : choice(c_r, 1, T) \vee choice(c_r, 2, T).$$
$$\forall T, I, J : (choice(c_r, I, T) \wedge choice(c_r, J, T)) \rightarrow I = J.$$

Depending on the values of the *choice* atoms, the definition at a particular time t is equivalent to either

$$causes(t, f_1) \leftarrow happens(a, t).$$

or

$$causes(t, f_2) \leftarrow happens(a, t).$$

Note that *choice* has three parameters: the first is to distinguish between rules (for each nondeterministic rule separate choices are required), the

second to distinguish between choices for one rule, and the third is a time parameter (as the effect of a nondeterministic action may vary at different time points). Also of importance is that *choice* is an open predicate for which no definition and only the above FOL axioms exist. Thus *choice* atoms can take on all possible values such that for each c_r and T, exactly one atom $choice(c_r, I, T)$ is true. For each (Ith) disjunct in a nondeterministic rule r, one separate atom $choice(c_r, I, T)$ is used. If a disjunct is itself a conjunction, clauses with equal bodies are generated for each conjunct.

We can formalise the above method as follows: a rule $r =$

$$a \textbf{ causes } \bigwedge_{j=1\ldots n_1} l_{1,j} \mid \ldots \mid \bigwedge_{j=1\ldots n_m} l_{m,j} \textbf{ if } F$$

is mapped to the set of clauses

$$\bigcup_{i=1\ldots m, j=1\ldots n_i} \{causes(T, l_{i,j}) \leftarrow happens(a,T), holds(F,T), choice(c_r, i, T).\}$$

and the FOL axioms

$$\forall T : choice(c_r, 1, T) \vee \ldots \vee choice(c_r, m, T).$$
$$\forall T, I, J : (choice(c_r, I, T) \wedge choice(c_r, J, T)) \rightarrow I = J.$$

with c_r a constant not occurring elsewhere in the theory. A rule $r' =$

$$\textbf{initiating } F' \textbf{ causes } \bigwedge_{j=1\ldots n_1} l_{1,j} \mid \ldots \mid \bigwedge_{j=1\ldots n_m} l_{m,j} \textbf{ if } F$$

is mapped to the set of clauses

$$\bigcup_{i=1\ldots m, j=1\ldots n_i} \{causes(T, l_{i,j}) \leftarrow \bigwedge_{l \in L_1} causes(T, l), \bigwedge_{l' \in L_2} (holds(l', T), \neg causes(T, \overline{l'})), \neg holds(F', T), holds(F, T), choice(c_{r'}, i, T). \mid L_1 \cup L_2 \text{ is supporting set of } F'\}$$

and the FOL axioms

$$\forall T : choice(c_{r'}, 1, T) \vee \ldots \vee choice(c_{r'}, m, T).$$
$$\forall T, I, J : (choice(c_{r'}, I, T) \wedge choice(c_{r'}, J, T)) \rightarrow I = J.$$

with $c_{r'}$ a constant not occurring elsewhere in the theory.

The interaction between the different forms of complexity makes for a harder to read notation, so let us give another example.

Consider the rule

$$\textbf{initiating } f_1 \vee f_2 \textbf{ causes } (g_1 \wedge g_2) \mid h \mid true \textbf{ if } true$$

This rule is mapped to the clauses

$$\begin{array}{lll}
causes(T, g_1) & \leftarrow & causes(T, f_1), \neg holds(f_1 \vee f_2), choice(c, 1, T). \\
causes(T, g_1) & \leftarrow & causes(T, f_2), \neg holds(f_1 \vee f_2), choice(c, 1, T). \\
causes(T, g_2) & \leftarrow & causes(T, f_1), \neg holds(f_1 \vee f_2), choice(c, 1, T). \\
causes(T, g_2) & \leftarrow & causes(T, f_2), \neg holds(f_1 \vee f_2), choice(c, 1, T). \\
causes(T, h) & \leftarrow & causes(T, f_1), \neg holds(f_1 \vee f_2), choice(c, 2, T). \\
causes(T, h) & \leftarrow & causes(T, f_2), \neg holds(f_1 \vee f_2), choice(c, 2, T). \\
causes(T, true) & \leftarrow & causes(T, f_1), \neg holds(f_1 \vee f_2), choice(c, 3, T). \\
causes(T, true) & \leftarrow & causes(T, f_2), \neg holds(f_1 \vee f_2), choice(c, 3, T).
\end{array}$$

and the FOL axioms

$$\begin{array}{l}
\forall T : choice(c, 1, T) \vee choice(c, 2, T) \vee choice(c, 3, T). \\
\forall T, I, J : (choice(c, I, T) \wedge choice(c, J, T)) \rightarrow I = J.
\end{array}$$

where the last two clauses can in fact be omitted (which is not done in the formalisation for simplicity reasons). The number of generated clauses is equal to the number of literals in the head times the number of supporting sets of the body of the derived effect rule.

We now prove that the extended mapping is correct, which is a slightly more complicated task than in the deterministic case. First, observe that with each $\mathcal{ER}$-interpretation I corresponds a class of OLP-interpretations which assigns the same value as I to all instances of predicates occurring in the $\mathcal{ER}$-theory, and an arbitrary value to each instance of the choice predicate. The correctness criterion we need to prove is then the following: an $\mathcal{ER}$-interpretation is a model of the $\mathcal{ER}$-theory if and only if at least one of its corresponding OLP-interpretations is a model of the OLP theory. More precisely this reads:

Theorem 7.6.1 *An $\mathcal{ER}$-interpretation is a model of Π_p and a model of at least one of the groundings of Π_e if and only if at least one of its corresponding OLP-interpretations is a model of the OLP theory.*

We first prove an important lemma.

Definition 7.6.3 *Given an $\mathcal{ER}$-interpretation I and an interpretation J_G of the choice predicate which satisfies the FOL axioms for that predicate. $J_{I,G}$ is the OLP-interpretation which corresponds to I (i.e. assigns the value corresponding to I to each atom corresponding to an $\mathcal{ER}$-atom) and which coincides with J_G on the choice predicate.*

Lemma 7.6.1 *To each grounding G of Π_e corresponds an interpretation J_G of the choice predicate such that an $\mathcal{ER}$-interpretation I is a model of Π_p and G if and only if $J_{I,G}$ is a model of the OLP theory.*

Proof:
To prove the lemma, we determine an interpretation of the *choice* predicate which allows us to use the proof of the deterministic case given in section 7.5. For a particular $J_{I,G}$, it is easy to see that all steps of the proof apply without modification, except for one of the inductive steps, i.e. the proof that at any time t, $causes(t,l) \leftrightarrow Causes(t,l)$ follows from $holds(F,t) \leftrightarrow \mathcal{H}o(F,t)$. For this to be valid it is required that G is equivalent to the definition of *causes*. Hence, if we can find an interpretation J_G which makes the OLP-definition of *causes* equivalent to the given grounding G of Π_e, it follows that I is a model of $G \cup \Pi_p$ if and only if $J_{I,G}$ is a model of the OLP theory.

Determining an appropriate interpretation of *choice* is not difficult: it follows from the definition of groundings and the construction of the mapping that, for each nondeterministic rule r and each time point t, the set of clauses in the mapping of r is equivalent to the kth grounding of r (i.e. the grounding corresponding to r's kth disjunctive component), if $choice(c_r, k, t)$ is true and all other $choice(c_r, j, t)$ are false. Indeed, as indicated before, given these values for $choice(c_r, i, t)$, all clauses corresponding to other groundings than the kth one have a trivially false body at t whereas the clauses corresponding to the kth grounding are the clauses in the grounding of the kth disjunctive component with an additional true atom in the body.

So, given a grounding G corresponding at each time point t and for each rule r to the $k_{r,t}$th disjunct of r, if we choose J_G such that $choice(c_r, i, t)$ is false if $i \neq k_{r,t}$ and true if $i = k_{r,t}$, then I is a model of $G \cup \Pi_p$ if and only if $J_{I,G}$ is a model of the OLP theory. Moreover this choice satisfies the given FOL axioms on *choice*. This proves the lemma. □

Proof:
(of the theorem) The theorem follows immediately from the lemma if each interpretation J of the *choice* predicate corresponds to a grounding G. This is indeed the case: in each J exactly one atom, say $choice(c_r, i_{r,t}, t)$, is true for each r and t, due to the FOL axioms. By the construction of J_G, J then corresponds to the grounding G obtained by taking the $i_{r,t}$th disjunct of each r at each t. □

The above reasoning proves the correctness of the proposed mapping. We can then turn our attention to the relation between nondeterministic rules in $\mathcal{ER}$ and the causal rules introduced in [109].

7.7 $\mathcal{ER}$ Compared with the Approach of Thielscher

In this section we make a detailed comparison of $\mathcal{ER}$ derived effect rules with Thielscher's approach to the ramification problem ([109]). The reason for this is twofold: first of all Thielscher's approach is one of the most recent ones, has been compared by Thielscher with a good deal of other recent approaches, and looks reasonably similar to ours. Therefore we think it is worth analysing the correspondence in detail. A second reason is that Thielscher has introduced the use of influence information for deriving certain causal rules from state constraints, which has struck us as a very interesting idea. Further in this chapter we will propose a different approach in $\mathcal{ER}$ and show how it relates to Thielscher's.

In what follows we will distinguish between *causal rules* (the constructs used by Thielscher) and *derived effect rules* (their counterpart in $\mathcal{ER}$). We will show a close correspondence between Thielscher's causal rules and non-deterministic derived effect rules.

In [109], causal rules have the form

$$e \textbf{ causes } l \textbf{ if } F$$

where e and l are fluent literals and F a fluent formula. Intuitively, the semantics of such a rule is that strong initiation of e may cause weak initiation of l if F holds.[30] More precisely:

Definition 7.7.1 (concepts related to causal rules)
Given a state S and a set of effects E (i.e. a set of fluent literals known to be strongly initiated), the rule

$$e \textbf{ causes } l \textbf{ if } F$$

is applicable in (S, E) iff $e \in E$ and $S \models F \wedge e \wedge \bar{l}$.[31]
Applying this rule results in a new state $(S \setminus \{\bar{e}\}) \cup \{e\}$ and a new set of effects $E \cup \{e\}$.[32]
The successor states after the execution of an action A in a state S are obtained by applying sequences of applicable causal rules to the pair (S^*, E), where E is the set of direct effects of A and $S^* = (S \setminus \overline{E}) \cup E$, i.e. the state obtained after applying the direct effects E to S. S' is a successor state of S after action A iff a pair (S', E') is obtained from (S^*, E) after any

[30] We say "may cause" rather than "causes" since rules never need to be applied in Thielscher's approach.
[31] The condition $S \models \neg l$ guarantees that all computed effects are strong initiations.
[32] Observe that E may contain a subset of the form $\{e, \neg e\}$.

sequence of rule applications and S' satisfies all of the state constraints. In the sequel, we call the above E' the justifying set of effects of S'.

It is important to note that any sequence of applicable rules which leads to a state satisfying the state constraints, is valid. In particular, such a sequence does not need to contain *all* applicable rules, hence the intuitive reading that rules "may" cause effects. As a result, nondeterministic ramifications are obtained, as will be apparent in the comparison below.

There is a partial correspondence between Thielscher's causal rules and nondeterministic derived effect rules in $\mathcal{ER}$. Let us first look only at causal rules of the form

$$a \textbf{ causes } b$$

with a and b fluent literals. Such a rule states that a strong initiation of a *justifies* the initiation of b, i.e. it may result in the initiation of b, but does not necessarily do so (since a rule is never forced to be applied). This corresponds at first sight to our nondeterministic derived effect rule

$$\textbf{initiating } a \textbf{ causes } b \mid true \textbf{ if } true.$$

The rules are indeed equivalent under the condition of *initiation consistency* (a fluent cannot at the same time be initiated and terminated).[33] To make this more precise, we introduce the following notation.

Definition 7.7.2 (initiation consistency) *A set of effects E is* initiation consistent *iff it does not contain a subset of the form $\{f, \neg f\}$.*

Definition 7.7.3 (applicability/justification for causal rules)
A fluent literal a is strongly initiated in (S, E) iff $a \in E$ and E is initiation consistent. Then, it follows from the above definition that a causal rule a **causes** *b is applicable in (S, E) iff a is strongly initiated in (S, E). We say the rule then justifies initiation of b, and moreover justifies strong initiation of b iff $b \notin S$.*

Definition 7.7.4 (applicability/justification/(candidate) successor state in $\mathcal{ER}$)
A derived effect rule **initiating** a **causes** b **if** *true or a nondeterministic*

[33] This condition is not imposed by Thielscher since in his approach change propagations are assumed to happen consecutively and to take a very small amount of time. Therefore there is no problem with initiations and terminations of fluents in the same batch of effects. However, we prefer not to adopt such a treatment of very small delays and for now we consider all propagations to be simultaneous and instantaneous, in which case initiation consistency is an essential condition. We return to the issue of delays in much detail in section 7.9.

derived effect rule **initiating** a **causes** b | *true* **if** *true is applicable at* t *iff* a *is strongly initiated at* t. *The rule then justifies initiation of* b, *and justifies strong initiation of* b *iff* b *does not hold at* t. *Applying this rule to a set* E *of strongly initiated literals at* t *yields the set* $E \cup \{b\}$.
A set of fluent literals S' *is a candidate successor state of* S *after* A *according to a set of effect rules* Π_e *iff there is a grounding* $\mathcal{D}_{init}$ *of* Π_e *for which it follows from* $\mathcal{H}a(A,t) \wedge \forall l \in S : \mathcal{H}o(l,t)$ *that* $S' \setminus S = \{l \mid I_{\mathcal{D}_{init}}(\mathbf{Init}(t,l')) = \mathbf{t}$ *and* $S' \cap S = \{l' \in S \mid I_{\mathcal{D}_{init}}(\mathbf{Init}(t,\overline{l'})) = \mathbf{f}$. *Intuitively a candidate successor state is a set of literals that can be true immediately after the action according to the effect rules.*
A candidate successor state S' *is a successor state iff it satisfies all the state constraints.*

Note that the above definitions of applicability deviate from Thielscher's in that we don't impose $S \models a \wedge \neg b$. However, from the construction of (S,E) pairs, it follows that $E \subseteq S$ if E is initiation consistent. Hence $S \models a$ is trivially satisfied if $a \in E$. As for b, we say a rule is applicable even if b already holds. However, the rule then only justifies weak, rather than strong initiation of b.

We then have the following results:

Lemma 7.7.1 *Given a state* S *at time* t *and an initiation consistent set* E *of strongly initiated literals at* t, *the causal rule*

a **causes** b

justifies strong initiation of a literal $b \notin (E \cup \overline{E})$ *in* $((S \setminus \overline{E}) \cup E, E)$ *if and only if the derived effect rule*

initiating a **causes** b **if** *true*

justifies the same strong initiation at t.

Proof:
Either rule is applicable if and only if a is strongly initiated. In that case, the causal rule justifies strong initiation of b iff $b \notin ((S \setminus \overline{E}) \cup E)$. Since $b \notin (E \cup \overline{E})$, this is equivalent to the condition $b \notin S$. On the other hand, the derived effect rule justifies strong initiation of b at t iff b does not hold at t. Since S is the state at t, this is also equivalent to $b \notin S$. □

Lemma 7.7.2 *Given a state* S *at time* t *and an initiation consistent set* E *of strongly initiated literals at* t, *applying the causal rule*

a **causes** b

yields an initiation consistent set of strong initiations E' if and only if applying the derived effect rule

initiating a **causes** b **if** *true*

to E at t yields the same initiation consistent set E'.

Proof:
There are three cases. If $b \in E$ then $E = E'$. If $b \in \overline{E}$ then E' is not initiation consistent. If $b \notin (E \cup \overline{E})$, either both rules justify strong initiation of b, in which case $E' = E \cup \{b\}$, or neither justifies b, in which case $E' = E$. □

Lemma 7.7.3 *Given a state S and set of effects E, and a causal rule mapping (S, E) to (S', E'). If E' is initiation consistent, any causal rule applicable in (S, E) is also applicable in (S', E'). Likewise, any nondeterministic derived effect rule applicable at t given E is also applicable at t given E' if E' is initiation consistent.*

Proof:
The lemma follows immediately from the observation that $E \subseteq E'$. □

Lemma 7.7.4 *Given a state S at time t, an action A with direct effects E occurring at t, a set of causal rules $\{C_1, \ldots, C_n\}$ and a corresponding set of derived effect rules $\{D_1, \ldots, D_n\}$.*

1. *If (S', E') is obtained by applying a sequence of rules $(C_{s_1}, \ldots, C_{s_m})$ to $((S \setminus \overline{E}) \cup E, E)$ and S' satisfies the state constraints, then*
   (a) *If E' is initiation consistent, then S' is the successor state of S according to the set of derived effect rules $\{D_{s_1}, \ldots, D_{s_m}\}$.*
   (b) *If E' is not initiation consistent, S has no successor state after A according to $\{D_{s_1}, \ldots, D_{s_m}\}$.*
2. *If S' is the successor state of S after A according to a subset of the derived effect rules $\{D_1, \ldots, D_n\}$, it is a successor state according to the corresponding subset of causal rules and its justifying set of effects E' is initiation consistent.*

Proof:

1. From the previous lemmas it follows that applying the sequence of rules $(C_{s_1}, \ldots, C_{s_m})$ to $(S_0, E_0) = ((S \setminus \overline{E}) \cup E, E)$ yields a sequence

of (state,effect) pairs $((S_0, E_0), \ldots, (S_m, E_m))$ such that D_{t_i} is applicable to (S_{i-1}, E_{i-1}) and yields (S_i, E_i). Since each derived effect rule remains applicable in all $(S_j, E_j), i < j$ if it is applicable in (S_i, E_i), all derived effect rules are applicable in (S_m, E_m), i.e. their conditions are satisfied. Since all derived effect rules are applied in the construction and the set of effects grows monotonically, the heads of all rules are also satisfied in (S_m, E_m).

Now, assume S' is the candidate successor state of S according to the derived effect rules. Note that these rules form a definite definition, and there exists exactly one such S'. This S' is a successor state of S iff the initiation consistency condition in $\mathcal{ER}$ is satisfied and S' satisfies the state constraints.

(a) Assume E_m is initiation consistent. We then show that $S' = S_m$. Since only the heads of applicable rules are added to E and the number of rule applications is finite, each initiation of a literal in E_m has a true finite proof tree, hence all literals in E_m are true in S'. The initiation of any literal not in $E_m \cup S$ only has a false proof tree, since if it had been the head of any applicable rule, it would have been added to E. Therefore all literals not in $E_m \cup S$, i.e. all literals in $\overline{S} \setminus E_m$, are false in S'. Since S is a state, one of each pair $(f, \neg f)$ is true in S. Hence the true literals in S' are exactly those in $(S \setminus \overline{E_m}) \cup E_m$, which is equal to S_m since E_m is initiation consistent. By the construction $S_m = S'$ satisfies all the state constraints, so S' is a successor state of S.

(b) Assume E_m is not initiation consistent. Then S' is not a successor state of S as the initiations leading to it violate the initiation consistency condition in $\mathcal{ER}$. Since S' is the unique candidate successor state for S, S has no successor state.

2. Assume S' is the successor state of S after A according to the direct effect rules for A and the derived effect rules $\{D_{t_1}, \ldots, D_{t_k}\}$. Starting from $(S_0, E_0) = ((S \setminus \overline{E}) \cup E, E)$, we can build a sequence $((S_0, E_0), \ldots, (S_m, E_m))$ by randomly applying one applicable causal rule from the set $\{C_{t_1}, \ldots, C_{t_k}\}$ in each step until no more rules apply. Assume we order the C_{t_j} and D_{t_j} such that C_{t_i} is the rule applied in step i. E_m may now be initiation consistent or not.

   Assume then E_m is initiation consistent. S_m is the successor state according to $\{D_{t_1}, \ldots, D_{t_m}\}$. The rules $C_{t_{m+1}}, \ldots, C_{t_k}$ are not applicable in (S_m, E_m), so neither are $D_{t_{m+1}}, \ldots, D_{t_k}$. Consider then adding these rules to the definition $\{D_{t_1}, \ldots, D_{t_m}\}$. Since the rules $D_{t_{m+1}}, \ldots, D_{t_k}$ are not applicable given E_m, their bodies are not in

E_m, so the additional proof trees they give rise to can not contain true leaves. Hence, only finitely failed or infinite proof trees are added to the set of proof trees. Since such proof trees yield false as a truth value, the value of the best proof tree of each literal is unchanged. So S_m is also the successor state of S after A according to $\{D_{t_1}, \ldots, D_{t_m}\}$, i.e. $S_m = S'$.

On the other hand, if E_m is initiation inconsistent, the candidate successor state $S' = S_m$ violates the initiation consistency condition. The addition of $\{D_{t_{m+1}}, \ldots, D_{t_k}\}$ to the definition results only in additional proof trees for some effects, so any effect with a true proof tree also has this true proof tree according to the extended definition. Hence, the set of effects certainly contains E_m and therefore is initiation inconsistent. So according to $\{D_{t_1}, \ldots, D_{t_k}\}$ there is no successor state to S.

□

Theorem 7.7.1 *Given a state S at time t, an action A with direct effects E occurring at t, a set of causal rules $\{C_1, \ldots, C_n\}$ where C_i = a_i* **causes** *b_i and a corresponding set of nondeterministic derived effect rules $\{ND_1, \ldots, ND_n\}$ where ND_i =* **initiating** *a_i* **causes** *b_i | true* **if** *true. S' is a successor state for S after A according to $\{ND_1, \ldots, ND_n\}$ iff S' is a successor state of S after A according to $\{C_1, \ldots, C_n\}$ and the justifying set of effects E' of S' is initiation consistent.*

Proof:
S' is a candidate successor state of S according to $\{C_1, \ldots, C_n\}$ if it is obtained after applying any sequence of applicable causal rules. This is equivalent to the condition that S' is obtained after application of a maximal sequence of causal rules of any subset of $\{C_1, \ldots, C_n\}$ (since each sequence is a maximal sequence in any subset of $\{C_1, \ldots, C_n\}$ containing all the applied rules and none of the applicable unapplied rules, and since vice versa any maximal sequence in a subset is also a sequence in that subset and hence a sequence in the entire set). An S' satisfying the above condition is a successor state for S iff it satisfies all of the state constraints.

In $\mathcal{ER}$ the candidate successor states of S according to $\{ND_1, \ldots, ND_n\}$ are those obtained by any grounding of this definition. Consider such a grounding: by definition, for a rule **initiating** a_i **causes** b_i | *true* **if** *true*, for each particular time point t the grounding of this rule is equivalent to the grounding of D_i = **initiating** a_i **causes** b_i **if** *true* or to the grounding of the trivially satisfied rule T_i = **initiating** a_i **causes** *true* **if** *true*. Hence, S'' is a successor state of S according to $\{ND_1, \ldots, ND_n\}$ if and only if it is a

candidate successor state of S according to any definition $\{D_{i_1}, \ldots, D_{i_k}\} \cup \{T_{j_1}, \ldots, T_{j_l}\}$ such that $(\{i_1 \ldots i_k\}, \{j_1 \ldots j_l\})$ is a partition of $\{1 \ldots n\}$. Since the rules T_j only result in proof trees for *true*, omitting them from the definition does not influence the candidate successor state. Hence S'' is a candidate successor state of S according to $\{ND_1, \ldots, ND_n\}$ iff it is the candidate successor state according to any subset $\{D_{i_1}, \ldots, D_{i_k}\}$ of $\{D_1, \ldots, D_n\}$. It is a successor state of S iff in addition it satisfies the state constraints and the set of effects leading to it is initiation consistent.

From the previous lemma it follows that the conditions on S' and S'' are equivalent if E' is initiation consistent, which proves the theorem. □

The above comparison shows that Thielscher's causal rules a **causes** b have the same semantics as $\mathcal{ER}$ rules **initiating** a **causes** $b \mid true$ **if** $true$ under the condition of initiation consistency.

The situation is more complicated for causal rules of the form

a **causes** b **if** F

which, roughly speaking, say that strong initiation of a causes strong initiation of b provided that F holds. Such rules do *not* correspond to our rules

initiating a **causes** $b \mid true$ **if** F.

We will show that in $\mathcal{ER}$ there is no equivalent for this complex type of causal rule. Closest to it is the combination of the two derived effect rules

initiating a **causes** $b \mid true$ **if** F
initiating $a \wedge F$ **causes** $b \mid true$ **if** $\neg a$

The first rule states that b may be initiated if a is strongly initiated, provided that F holds in the starting state S for this set of effects. The second rule states the same relation between changes in a and b, but now under the condition that F holds in the resulting state S' rather than the starting state (initiating $a \wedge F$ given that a was false, is equivalent to strongly initiating a while making sure that F becomes or stays true). Yet these two rules do not cover all the cases in which Thielscher's causal rule is applicable.

This is due to the fact that in Thielscher's approach, rules are applied consecutively, giving rise to a set of intermediate states. A rule a **causes** b **if** F can be applied at a certain point in such a sequence if F holds in the appropriate intermediate state. Thielscher shows some examples in which a fluent holds in one of the intermediate states, but neither in the starting nor the resulting state. In such a case, none of our derived effect rules would be applicable, but Thielscher's causal rule would. If F is a fluent, this difference is eliminated by the initiation consistency condition, which prevents fluents from being initiated and terminated in the

same batch of effects. However, as F can be a general formula, to obtain equivalence the initiation consistency condition would have to be extended in a way which is no longer reasonable: if no formula can be initiated and terminated in one batch of effects, we cannot allow even two consecutive effects. Indeed, assume we initiate a and b consecutively, then for example $a \wedge \neg b$ and $\neg a \wedge b$ are initiated and terminated in the same batch of effects, which is in no way problematic or counterintuitive.

For causal rules of this more complex form, we have no exact equivalent in $\mathcal{ER}$. The best approximation is given by two rules, as indicated above. One consequence is that any approach to using influence information in $\mathcal{ER}$ will certainly deviate from Thielscher's. In the next section we present such a method and compare it to Thielscher's approach.

7.8 Influence Information

As we have extensively argued, in our approach ramifications are seen as manifestations of effect propagations. In other words, we assume that an expert modeling a dynamic domain models the effects existing in that domain and describes how these effects propagate. In $\mathcal{ER}$ these propagations are represented by derived effect rules. However, in the literature state constraints have always been used in a high level description of dynamic domains: these state constraints are easily observable, and as they can arise as a result of particular combinations of effect propagations, they are often strongly related to these propagations.

For this reason, Thielscher ([109]) proposes a method for automatically deriving causal rules representing ramifications from state constraints, using influence information. The idea is that if two fluents f and g occur in the same state constraint, and f influences g, then a change in f which results in a violation of the constraint may cause an appropriate change in g which restores the validity of the constraint. The appropriate causal rules describing these change propagations are then derived.

Thielscher's causal rules thus have the explicit task of restoring the validity of state constraints when they are violated. Hence they are not descriptions of known effect propagations like derived effect rules in $\mathcal{ER}$: causal rules only need to be applied if there are violated state constraints and if they can solve that problem. This is reflected in the fact that they map to nondeterministic derived effect rules in $\mathcal{ER}$: they describe potential effect propagations that may occur if they are required.

Despite the different point of view on ramifications, the idea of automatically deriving ramification rules from state constraints and influence information also deserves consideration in $\mathcal{ER}$. A formal characterisation

of the relation between state constraints, influence information and derived effect rules can be useful to help an expert derive the precise effect propagation rules starting from a high-level description consisting only of state constraints and vague influence information.

Of course such an approach has limitations. First of all, as we have shown in section 7.2, not all effect propagations are related to state constraints. Hence it is not possible in general to generate all effect propagation rules using state constraints and influence information. Second, even in the case of state constraint related ramifications the addition of influence information may still leave the ramifications underspecified, as we will show. Finally, determining whether a set of rules is the intended set for given constraints and influence information is not trivial: the problem at hand shows some similarities to the tasks of machine learning and intentional database updating.

Despite these limitations, the above argument still stands. Therefore we provide in this section a method for generating a set of derived effect rules corresponding to given state constraints and influence information. In addition, the rule sets we generate give an indication of when the effect propagations are underspecified, thus warning the user that the generated rules are based on insufficient information and may require further attention. In this respect the method should be seen as a tool helping the user and not as a fully independent answer to the ramification problem. Moreover the examples should show that such an independent answer is not feasible, despite approximations by both Thielscher and ourselves.

7.8.1 Generating Effect Propagation Rules

Our goal is to generate derived effect rules from a given set of state constraints and a given influence relation $Infl$ which is a binary relation on fluents: $(f, g) \in Infl$ if f can influence g. These derived effect rules should guarantee that whenever f is modified such that a particular state constraint containing f and g would be violated, an appropriate modification of g can occur which restores or helps to restore the validity of the constraint. On the other hand, indirect effects should also only be allowed if they satisfy the above condition.

We assume that the set of state constraints is written in conjunctive normal form, i.e. a conjunction of constraints where each constraint is a disjunction of fluent literals. We use the convention that in a constraint $l_1 \vee \ldots \vee l_m$, each l_j is either f_j or $\neg f_j$ for a particular fluent f_j.

Since the state constraints and influence information may not provide sufficient information to determine the intended effect rules, it is not possible in general to give necessary and sufficient conditions for a set of effect

rules to be correct. However, based on the intuitions about influence information sketched above, we propose the following correctness criterion, which is a necessary condition on the sets of effects that can be generated by the effect rules.

Definition 7.8.1 (weak correctness criterion)
A correct set of effects E with respect to given state constraints and influence information and a given starting state S, must satisfy the following conditions:

- *For any state constraint C, $(S \setminus \overline{E}) \cup E \models C$*
- *Each effect in E is either*
  - *a direct effect, or*
  - *an indirect effect l_k such that for $E' = E \setminus \{l_k\}$ there exists some $C = \bigvee_{i=1 \ldots n} l_i$ and some l_j $(1 \leq j, k \leq n)$ such that $(f_j, f_k) \in Infl$, $\overline{l_j} \in E'$, and for some $E'' \subseteq E'$ with $\overline{l_j} \in E''$, $S \setminus \overline{E''} \cup E'' \not\models C$*

In other words, resulting states must satisfy all state constraints, and each effect generated by the effect rules must be justified by some state constraint which would be violated due to other occurring effects and by the appropriate influence information. A stronger variant of the correctness criterion is obtained by imposing in addition that $E' = E''$:

Definition 7.8.2 (strong correctness criterion)

- *For any state constraint C, $S \setminus \overline{E} \cup E \models C$*
- *Each effect in E is either*
  - *a direct effect, or*
  - *an indirect effect l_k such that for $E' = E \setminus \{l_k\}$ there exists some $C = \bigvee_{i=1 \ldots n} l_i$ and some l_j $(1 \leq j, k \leq n)$ such that $(f_j, f_k) \in Infl$, $\overline{l_j} \in E'$, and $S \setminus \overline{E'} \cup E' \not\models C$*

This variant requires that each effect only takes place if the state constraint it intends to restore would be violated by the combination of *all* other effects. This adds a strong minimality condition to the weak correctness criterion: if there are two ways to restore the validity of a state constraint, only one of them should be adopted, never both. It can be argued that the strong version of the correctness criterion is to be preferred, and in fact

the rules generated by our approach satisfy the strong criterion. However we will show below that due to possible underspecification a violation of the strong criterion (which for example occurs in Thielscher's approach) is sometimes acceptable.

Our approach is modular, in that the derived effect rules are generated for each state constraint independently. First we study the case in which effect propagations follow unambiguously from a state constraint C and the influence information. It is easy to see that this is always the case when in C only one of the fluents, say f, can be influenced: then the only possible indirect effect is a modification of f. Hence, if such a modification is allowed by the influence information (i.e. if one of the fluents influencing f is changed) and if it would restore the constraint's validity, then f should always be changed: otherwise the state constraint would remain violated.

This propagation can be enforced as follows: for each state constraint $l_1 \vee \ldots \vee l_m$ (with each l_j either f_j or $\neg f_j$ for a particular fluent f_j), which contains at most one l_k such that f_k can be influenced by any other fluent in the language, generate the derived effect rule

$$\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } l_k \textbf{ if } \bigvee_{f_i \in I} l_i$$

where I is the set of fluents in the constraint that can influence f_k.

In this rule, the body $\bigwedge_{t=1\ldots m, t\neq k} \overline{l_t}$ is the negation of the state constraint with l_k left out: the formula denotes that a necessary condition for l_k to be caused is that in the resulting state, the state constraint is violated unless l_k is true. The condition of the rule, $\bigvee_{f_i \in I} l_i$, denotes that at least one of the l_i which can influence l_k is true in the starting state, and hence is changed, so that its effect can propagate.

This case covers a very large class of applications, including the suitcase, table and flying turkey examples presented in earlier sections and nearly all examples studied in the literature. For example in the suitcase domain the influence information is that l_1 and l_2 may influence *open*, so certainly in any state constraint at most one fluent (*open*) can be influenced. The state constraint in disjunctive form in this example is $\neg l_1 \vee \neg l_2 \vee open$, which leads to the derived effect rule

$$\textbf{initiating } l_1 \wedge l_2 \textbf{ causes } open \textbf{ if } \neg l_1 \vee \neg l_2$$

Note that this rule is syntactically different from the one we used when modeling the example earlier. Indeed, the above rule can be simplified to

$$\textbf{initiating } l_1 \wedge l_2 \textbf{ causes } open \textbf{ if } true$$

since $l_1 \wedge l_2$ can be initiated only if its negation is true. This simplification can be generalised: if the one fluent in a constraint which can be influenced is influenced by all other fluents in the constraint, we can generate the derived effect rule

$$\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } l_k \textbf{ if } true$$

which is equivalent with the one generated using the general method.

Next we consider the case of constraints in which multiple fluents can be influenced. In general there is no unique set of indirect effects which can restore the constraint's validity in this case: the problem is underspecified and/or involves nondeterminism.

This is very clear in the following example: consider the state constraint $a \rightarrow (b \vee c)$, in disjunctive form $\neg a \vee b \vee c$, with the influence information $\{(a,b),(a,c)\}$. Assuming that we start with a state in which all three fluents are false, and then initiate a, the constraint's validity can be restored by initiating either b or c, or both. Since there is no reason to prefer either b or c, the rules we generate must be nondeterministic. However it is not clear if only the minimal changes (initiating only b or only c) are to be considered, or also the change in both b and c. Adhering to the strong version of the correctness criterion given above, only the minimal solutions would be acceptable: if b is initiated there is no justification for c and vice versa. The weaker version of the correctness criterion allows for a change in both b and c, since both are justified by the change in a. Here we can argue there is a problem of underspecification.

To make the discussion more concrete, assume a, b and c are courses taught at a university. The constraint represents that b and c are prerequisites for a, and the influence information that if a student wants to follow course a but has not followed b or c, his/her subscription for a can be allowed by subscribing him/her in addition to course b or c. The given information does not specify which of b or c is to be preferred. Real students would probably not be happy with two additional subscriptions and prefer the minimal adaptations, but there may be exceptions to this rule. Also, usually one would talk to the student to determine which additional course (s)he prefers, and not assign one nondeterministically. In other words, more information is clearly required to determine the best course of action.

Given the above considerations, in our view the preferable general solution is to propose only minimal sets of changes but to clearly indicate that there is a problem of underspecification.

This is achieved by the following formalisation: if multiple fluents, say $f_1 \ldots f_s$, in one state constraint $l_1 \vee \ldots \vee l_m$ can be influenced by other

fluents in the theory, the nondeterministic derived effect rules

$$\textbf{initiating} \bigwedge_{i=1\ldots m, i\neq k} \overline{l_i} \textbf{ causes } l_k \mid \mathit{true} \textbf{ if } \bigvee_{f_i \in I_k} l_i$$

are generated for $1 \leq k \leq s$, where I_k is the set of fluents in the constraint that can influence f_k. In other words, for each l_k we obtain a rule like the one we obtained in the previous case, except for the fact that the effect of the rule is now optional: the rule is nondeterministic. Of course since state constraints need to be satisfied at all times, it is required that if the state constraint is violated by a set of direct effects at least one of the rules is applied. In the above example, the obtained effect rules are

$$\textbf{initiating } a \wedge \neg b \textbf{ causes } c \mid \mathit{true} \textbf{ if } \neg a$$
$$\textbf{initiating } a \wedge \neg c \textbf{ causes } b \mid \mathit{true} \textbf{ if } \neg a$$

At any particular time point, the semantics of the rules is given by one of four possible groundings, corresponding to the groundings of the following pairs of deterministic rules.

$$\textbf{initiating } a \wedge \neg b \textbf{ causes } \mathit{true} \textbf{ if } \neg a$$
$$\textbf{initiating } a \wedge \neg c \textbf{ causes } \mathit{true} \textbf{ if } \neg a$$

$$\textbf{initiating } a \wedge \neg b \textbf{ causes } c \textbf{ if } \neg a$$
$$\textbf{initiating } a \wedge \neg c \textbf{ causes } \mathit{true} \textbf{ if } \neg a$$

$$\textbf{initiating } a \wedge \neg b \textbf{ causes } \mathit{true} \textbf{ if } \neg a$$
$$\textbf{initiating } a \wedge \neg c \textbf{ causes } b \textbf{ if } \neg a$$

$$\textbf{initiating } a \wedge \neg b \textbf{ causes } c \textbf{ if } \neg a$$
$$\textbf{initiating } a \wedge \neg c \textbf{ causes } b \textbf{ if } \neg a$$

In the case where a is initiated while b and c are false, the first pair of rules does not lead to a state satisfying the state constraint, and can be disregarded. The second and third pairs of rules yield minimal changes restoring the validity of the constraint, initiating either b or c. The fourth pair of rules yields a truth value "u" for the initiations of both b and c, so this is not a correct constructive definition.

Recall that a set of initiations satisfies a nondeterministic definition if it is a model of one of the definition's groundings. Hence we find two sets of initiations consistent with both the definition and the state constraint: one in which b is initiated and one in which c is. These are precisely the minimal changes able to restore the state constraint's validity. However,

as we indicated we cannot be entirely certain if only the minimal changes should be considered: whether or not this is the case may be domain dependent, and the answer can in general not be given by state constraints and influence information alone. So we are not certain if the generated set of rules is the intended one, due to underspecification. Luckily, our proposal naturally incorporates a warning when such a problem of underspecification occurs: this warning is the presence of a bad definition in one of the groundings. Though this bad definition has no impact on the semantics (it is an additional grounding which yields no valid set of initiations), its presence warns the user that the problem may require further analysis. In this case the user may look into the details of the domain at hand, check which effect propagations are intended and decide to modify the generated rules if needed. By default the minimal change policy is maintained.

In the above example we had one fluent influencing two other fluents in the same constraint. The case in which two different fluents influence a third resp. fourth fluent in the same constraint, as in $\neg a \vee b \vee \neg c \vee d$ with influence information $\{(a, b), (c, d)\}$, is entirely similar: if a and c are initiated when b and d are false, the constraint may be restored by initiating either b or d. We again obtain two nondeterministic effect rules of which one grounding does nothing and yields a state violating the constraint, two groundings yield the minimal changes, and one grounding is a bad definition indicating a problematic specification.

A third possibility is that two fluents in a constraint C can be influenced, but only one of them by another fluent in C and the second one by a fluent outside C. For example, assume the constraints

$$\neg a \vee b \vee \neg c$$
$$\neg d \vee c$$

with influence information $\{(a, b), (d, c)\}$. For the first constraint, due to the influence of a on b, we obtain a rule

initiating $a \wedge c$ **causes** $b \mid true$ **if** $\neg a$

As should be expected, the influence of d on c does not lead to effect rules for the first constraint. It does however generate an — in this case deterministic, as only c can be influenced — rule for the second constraint:

initiating d **causes** c **if** $\neg d$

Assuming a state in which all fluents are false and a and d are initiated, the second rule will cause c to be initiated and as a result, since now $a \wedge c$ is initiated, the first rule may or may not initiate b depending on which of the

two groundings is considered. However, the case in which b is not initiated leads to a state violating the first constraint, so only the other grounding yields a model. Hence there is no nondeterminism in this case, even though the nondeterministic derived effect rule suggests there is.[34] Also, there is no grounding which yields a bad definition. Indeed there is no problem of underspecification in this example: the effect rules determine a unique set of effects which restores all state constraints.

We are now ready to prove that the proposed method generates rules that satisfy the correctness criterion given at the beginning of this section:

Theorem 7.8.1 *Any set of effects E obtained by applying to a particular valid $\mathcal{ER}$-state S a given set of direct effect rules and a set of derived effect rules generated from state constraints and influence information by the above method, and which yields a valid $\mathcal{ER}$-state S', satisfies the strong correctness criterion, i.e.*

1. *For any state constraint C, $S \setminus \overline{E} \cup E \models C$*
2. *Each effect in E is either*
   (a) *a direct effect, or*
   (b) *an indirect effect l_k such that for $E' = E \setminus \{l_k\}$ there exists some $C = \bigvee_{i=1..n} l_i$ and some l_j $(1 \leq j, k \leq n)$ such that $(f_j, f_k) \in Infl$, $\overline{l_j} \in E'$ and $S \setminus \overline{E'} \cup E' \not\models C$*

Proof:

1. Since $S' = S \setminus \overline{E} \cup E$ is a valid $\mathcal{ER}$-state, it entails all state constraints.
2. Any occurring effect l_k must occur in the head of a rule with a true body. This can either be a direct effect rule, or a derived effect rule obtained from the state constraints and influence information by our method. In the former case condition 2(a) is satisfied. In the latter case, the derived effect rule is of the form

$$\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } D_k \textbf{ if } \bigvee_{f_i \in I_k} l_i$$

where D_k is either l_k or $l_k \mid true$. This rule is necessarily obtained from the state constraint $C = l_1 \vee \ldots \vee l_m$ and the influence information that all fluents in I_k can influence f_k. For this rule to be

[34] One may wonder why we do not generate deterministic rules in this case. We show the difference in the next section when comparing deterministic and nondeterministic rules.

applicable it is required that some l_j that can influence f_k is true in S, and that $\bigwedge_{t=1\ldots m, t\neq k} \overline{l_t}$ is false in S'. Since $(S\backslash\overline{E'})\cup E' = (S'\backslash l_k)\cup\overline{l_k}$, the second of these conditions implies that $S\backslash\overline{E'}\cup E' \not\models C$. Also, the two conditions together imply that $\overline{l_j} \in E'$. Hence, condition $2(b)$ is satisfied.

□

7.8.2 A More Uniform Notation

The above method for dealing with influence information generates deterministic or nondeterministic effect rules depending on the number of fluents which can be influenced in each particular constraint. In this way we obtain clear deterministic rules when possible, and nondeterministic rules if there are multiple options. Moreover we obtain only one rule for each constraint and each fluent which can be influenced in it, thus keeping the number of rules relatively low.

Formally we can devise a simpler, equivalent characterisation of the generated effect rules, in which not only each constraint but also each item of influence information is dealt with independently, and moreover each (state constraint,influence item) pair is handled in exactly the same way. We achieve this by applying two simplifications.

Our first simplification consists of handling each (state constraint,influence item) pair independently: for each state constraint $l_1 \vee \ldots \vee l_m$ and for each influence information item (f_i, f_k), we generate the derived effect rule

$$\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } l_k \textbf{ if } l_i$$

if l_k is the only fluent in the constraint which can be influenced, and

$$\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } l_k \mid \mathit{true} \textbf{ if } l_i$$

otherwise.

The set of rules obtained in this way is equivalent to the one obtained by our above method:

Theorem 7.8.2 *For a particular state constraint $l_1 \vee \ldots \vee l_m$ and a set of fluents $I_k = \{f_i \mid (f_i, f_k) \in Infl\}$, the derived effect rule*

$$\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } l_k \textbf{ if } \bigvee_{f_i\in I_k} l_i$$

is equivalent to the set of derived effect rules

$$\{\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } l_k \textbf{ if } l_i \mid f_i \in I_k\}.$$

Likewise, the nondeterministic derived effect rule

$$\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } l_k \mid true \textbf{ if } \bigvee_{f_i\in I_k} l_i$$

is equivalent to the set of derived effect rules

$$\{\textbf{initiating} \bigwedge_{t=1\ldots m, t\neq k} \overline{l_t} \textbf{ causes } l_k \mid true \textbf{ if } l_i \mid f_i \in I_k\}.$$

The proof is as follows. First consider the deterministic case: the grounding of a derived effect rule

$$\textbf{initiating } F \textbf{ causes } l \textbf{ if } F'$$

is

$$\begin{aligned}\{\textbf{Causes}(t,l) \leftarrow &\textbf{Init}(t,S_i), \overline{\textbf{Init}(t,\overline{S_p})},\\ &\mathcal{H}o(S_p,t), \neg\mathcal{H}o(F,t), \mathcal{H}o(F',t)\\ \mid t\in \mathbf{T} &\text{ and } S_i \cup S_p \text{ is a supporting set of } F\}.\end{aligned}$$

Since the derived effect rules mentioned in the theorem differ only in their condition, each yields a set of clauses of the form

$$\begin{aligned}\textbf{Causes}(t,l_k) &\leftarrow B_1, \mathcal{H}o(F',t).\\ &\vdots\\ \textbf{Causes}(t,l_k) &\leftarrow B_n, \mathcal{H}o(F',t).\end{aligned}$$

with F' the condition of the particular rule. Hence, the theorem follows if we can prove that each clause

$$\textbf{Causes}(t,l_k) \leftarrow B_j, \mathcal{H}o(\bigvee_{f_i\in I_k} l_i, t).$$

is equivalent to the set of clauses

$$\{\textbf{Causes}(t,l_k) \leftarrow B_j, \mathcal{H}o(l_i,t). \mid f_i \in I_k\}$$

Now, the first clause is equivalent to

$$\textbf{Causes}(t,l_k) \leftarrow B_j.$$

if $\mathcal{H}o(\bigvee_{f_i \in I_k} l_i, t)$ is true, and to

$$\textbf{Causes}(t, l_k) \leftarrow \mathit{false}.$$

otherwise. On the other hand, the set of "primitive" clauses is equivalent to

$$\textbf{Causes}(t, l_k) \leftarrow B_j.$$

if one of the $\mathcal{H}o(l_i, t)$, and hence $\bigvee_{f_i \in I_k} \mathcal{H}o(l_i, t)$, is true, and to

$$\textbf{Causes}(t, l_k) \leftarrow \mathit{false}.$$

otherwise. The equivalence then follows from the fact that $\mathcal{H}o(\bigvee_{f_i \in I_k} l_i, t) = \bigvee_{f_i \in I_k} \mathcal{H}o(l_i, t)$.

The proof for nondeterministic rules is obtained by applying the above reasoning to each grounding independently. □

A second simplification eliminates the distinction between constraints in which only one fluent is influenced and the other constraints. We can always use the following unique rule for deriving effect rules, regardless of the number of influenced fluents: for each state constraint $l_1 \vee \ldots \vee l_m$ and for each influence information item (f_i, f_k), we generate the nondeterministic derived effect rule

$$\textbf{initiating} \bigwedge_{t=1\ldots m, t \neq k} \overline{l_t} \textbf{ causes } l_k \mid \mathit{true} \textbf{ if } l_i.$$

In other words, we can always generate nondeterministic rules. The resulting definition is equivalent with the one using deterministic rules for constraints in which only one fluent is influenced.

Theorem 7.8.3 *Given a state constraint $l_1 \ldots l_m$ in which only l_k can be influenced. The derived effect rule*

$$D = \textbf{initiating} \bigwedge_{t=1\ldots m, t \neq k} \overline{l_t} \textbf{ causes } l_k \textbf{ if } l_i.$$

is equivalent to, i.e. leads to the same successor states as, the nondeterministic derived effect rule

$$ND = \textbf{initiating} \bigwedge_{t=1\ldots m, t \neq k} \overline{l_t} \textbf{ causes } l_k \mid \mathit{true} \textbf{ if } l_i.$$

Proof:
ND has two groundings: one which corresponds to the grounding of D and

one which only contains a rule for *true* (which corresponds to an omission of D from the rule set). Hence, it suffices to prove that this second grounding does not yield successor states not generated by the first grounding. Turning this around, it is sufficient to prove that any successor state of the rule set without D is also a successor state of the rule set with D.

Now, assume S' is a successor state of the rule set without D. If in the starting state S the fluent l_i is false, then bodies of rules derived from D are always false, so D has no effect. Otherwise, first observe that D's addition cannot introduce new proof trees for any of the l_t, due to the fact that none of the l_t can be influenced. In other words, D cannot introduce cycles (in particular over negation), so the truth value of $\bigwedge_{t=1\ldots m, t\neq k} \overline{l_t}$ remains invariable with or without D. We then have the following cases. If $\bigwedge_{t=1\ldots m, t\neq k} \overline{l_t}$ is false in S', then D is not applicable so its addition has no effect. Otherwise, $\bigwedge_{t=1\ldots m, t\neq k} \overline{l_t}$ is true (undefined truth values are not possible in a state). In that case, if l_k is true in S', D has no additional effect, and if l_k is false in S', then S' is no successor state since the state constraint is violated. Hence, we find that the addition of D does not eliminate any successor states of the rule set without D, which due to the above reasoning ensures that D and ND are equivalent. □

One might expect that also for rules generated from constraints in which only one fluent can be influenced from within the constraint (but in which other fluents can be influenced from without the constraint, as in the last example of section 7.8.1), we would obtain a similar equivalence result between deterministic and nondeterministic rules. However, the following example shows that there is no equivalence in that case: take two constraints

$$a \vee c \vee d \qquad b \vee c \vee d$$

and influence information $\{(a, c), (b, d)\}$. In both constraints, one fluent can be influenced from within and one from without the constraint. The rules generated from these data would be

initiating $\neg a \wedge \neg d$ **causes** $c \mid true$ **if** a
initiating $\neg b \wedge \neg c$ **causes** $d \mid true$ **if** b

which for a starting state $\{a, b, \neg c, \neg d\}$ and direct effects $\neg a, \neg b$ would generate two valid successor states obtained by minimal changes, $\{\neg a, \neg b, c, \neg d\}$ and $\{\neg a, \neg b, \neg c, d\}$, and in addition a bad grounding indicating that there is insufficient information. On the other hand, the corresponding deterministic rules

initiating $\neg a \wedge \neg d$ **causes** c **if** a
initiating $\neg b \wedge \neg c$ **causes** d **if** b

would only yield a bad definition and no successor states at all.

We have now obtained a uniform and modular characterisation of the derived effect rules corresponding to state constraints and influence information, which by the above theorems is equivalent to our original proposal. This original proposal can be seen as a more intuitive approach, in which sets of constraints obtained by the uniform method are contracted into single constraints, and in which apparent but non-existing nondeterminism has been eliminated. One advantage of the uniform method is that it will help us establish a correspondence with the approach in [109].

7.8.3 Comparing our Method with Thielscher's

We study the relation between our approach and the one in [109]. Formally, the causal rules in Thielscher's approach are computed as follows: assume $D_1 \wedge \ldots \wedge D_n$ is the conjunctive normal form of the conjunction of all state constraints, and each $D_i = l_1 \vee \ldots \vee l_{m_i}$ with all l_j fluent literals f_j or $\neg f_j$. Then for each D_i and for each (f_j, f_k) in the influence relation with l_j and l_k in D_i, the causal rule

$$\overline{l_j} \textbf{ causes } l_k \textbf{ if } \bigwedge_{t=1\ldots m_i, t\neq j, t\neq k} \overline{l_t}$$

is generated.

For the same constraint and influence information, we generate the rule

$$\textbf{initiating } \bigwedge_{t=1\ldots m_i, t\neq k} \overline{l_t} \textbf{ causes } l_k \mid true \textbf{ if } l_j$$

Recall from the previous section that this derived effect rule, together with

$$\textbf{initiating } \overline{l_j} \textbf{ causes } l_k \mid true \textbf{ if } \bigwedge_{t=1\ldots m_i, t\neq k, t\neq j} \overline{l_t}$$

is the closest approximation in $\mathcal{ER}$ of the above causal rule. In other words, there is an immediately clear correspondence between the generated rules in both approaches, with our derived effect rules being strictly and considerably weaker than Thielscher's causal rules.

Consider a state constraint $a \vee b \vee \bigvee_i l_i$ with (a, b) in the influence relation. This gives rise to the causal rule

$$\neg a \textbf{ causes } b \textbf{ if } \bigwedge \overline{l_i}.$$

and to the derived effect rule

$$\textbf{initiating } \neg a \wedge \bigwedge_i \overline{l_i} \textbf{ causes } b \mid true \textbf{ if } a$$

The derived effect rule is applicable if $\neg a \wedge \bigwedge_i \overline{l_i}$ becomes true and a is strongly terminated. In that case b should be initiated for the constraint to remain satisfied. The corresponding causal rule is applicable in a sequence of causal rules if a is strongly terminated and $\bigwedge \overline{l_i}$ is true in some appropriate intermediate state. So the causal rule is applicable in a number of cases where the derived effect rule is not. In these cases however, the application of the rule is not required in order to restore the validity of the constraint it is derived from: the state constraint is only violated if all of its literals are false in the generated successor state, so if $\bigwedge \overline{l_i}$ holds in some intermediate (or the initial) state but not in the generated successor state, there is no need for b to be initiated. In other words, Thielscher's rules do not satisfy the strong correctness criterion: they sometimes generate unnecessary, non-minimal ramifications.

As an example, assume we have state constraints $a \vee \neg d$, $b \vee \neg d$ and $c \vee b \vee \neg a$. The influence relation is $\{(d,a),(d,b),(a,c)\}$. We then get the causal rules

d **causes** a
d **causes** b
a **causes** c **if** $\neg b$

Assume then we have a state $\{\neg a, \neg b, \neg c, \neg d\}$ and an action with direct effect d. Using the first two rules we find the successor state $\{a, b, \neg c, d\}$, which satisfies all of the state constraints. Also, using the first, third and second rule in that order we obtain a state $\{a, b, c, d\}$, which also satisfies the state constraints. Using the derived effect rules obtained from the state constraints,

initiating d **causes** $a \mid true$ **if** $\neg d$
initiating d **causes** $b \mid true$ **if** $\neg d$
initiating $a \wedge \neg b$ **causes** $c \mid true$ **if** $\neg a$

we only find the resulting state $\{a, b, \neg c, d\}$: the first two rules must be applied to restore the first two constraints, and then the condition of the third rule is not satisfied.

So in this case the causal rules give rise to an additional successor state which is reached by a non-minimal set of changes (the change in c is unnecessary). This is to be expected in general given the fact that the causal rules are applicable under weaker conditions than derived effect rules. More precisely we can prove that the set of valid successor states according to the $\mathcal{ER}$ theory is a subset of the set according to Thielscher's corresponding theory:

Theorem 7.8.4 *A successor state S' of a state S and action A with direct effects E according to the nondeterministic derived effect rules obtained*

from given influence information and state constraints using the above method, is also a successor state of S after A according to the set of causal rules derived using the same influence information and state constraints by Thielscher's method.

Proof:

It is sufficient to prove that in a particular successor state (which we will construct) S'' of S according to the causal rules, each literal true in S' is also true. Since no causal rule is ever forced to be applied, assume we will not apply any rules leading to the initiation of the negation of any literal in S'. So any literal already true in S or an intermediate state and still true in S' can be assumed to be true in S''. Likewise, we can assume any initiation atom false in the transition between S and S' to be false in the effects leading to S''. It remains to be proven that strongly initiated literals in S' can be made true in E'' and S''. This can be proven by induction on the depth of the best proof tree of each such strong initiation:

1. Assume the initiation, say of literal l, has a proof tree of depth 1. Then there is a direct effect rule A **causes** l **if** F in the theory with F true in S, and so an equivalent effect description is in Thielscher's formalisation: l is simply a direct effect of A, so it is true in $S_0 = (S \setminus \overline{E}) \cup E$ and all subsequent states.

2. Assume all initiations with a true proof tree of depth k have been derived by causal rules, leading to (S_{n_k}, E_{n_k}), and l has a proof tree of depth $k+1$. Then there is a rule

$$\textbf{initiating } l' \wedge \bigwedge_i l_i \textbf{ causes } l \mid true \textbf{ if } \overline{l'}$$

in Π_e such that each l_i is either true in S or has a proof tree of depth at most k, and l' also has a proof tree of depth at most k. There is also a corresponding causal rule

$$l' \textbf{ causes } l \textbf{ if } \bigwedge_i l_i$$

By the induction hypothesis, l' and each l_i which was not already true in S is true in S_{n_k}, and by the assumptions made above also the other l_i are still true in S_{n_k}. Hence the causal rule is applicable and l can be derived. Moreover by the above assumptions this will not cause any other applicable rule to become inapplicable. Assuming then $l_1 \ldots l_j$ are all of the literals with a best proof tree of depth $k+1$, their corresponding causal rules can be applied in any sequence

starting from (S_{n_k}, E_{n_k}), leading to $(S_{n_{k+1}}, E_{n_{k+1}})$. This proves the inductive step.

The theorem follows from this inductive argument. □

Related to this, we should mention that in some cases the derived effect rules generated by our method only lead to states in which some of the state constraints are still violated, whereas Thielscher's rules yield sets of indirect effects leading to a valid state. In other words, in some cases our rules lead to an action qualification where Thielscher's lead to ramifications. This occurs intuitively speaking when a particular constraint is restored in a non-minimal way, which has the side effect of restoring another violated constraint. An example is the following: assume the constraints

$$\neg b \vee a \qquad \neg b \vee e \qquad d \vee c$$
$$\neg a \vee c \qquad \neg a \vee \neg d \vee e$$

and the influence information

$$\{(b,a),(b,e),(a,d),(d,c)\}$$

From these data, we can derive the causal rules

1. b **causes** a
2. b **causes** e
3. a **causes** $\neg d$ **if** $\neg e$
4. $\neg d$ **causes** c

and on the other hand the nondeterministic derived effect rules

1. **initiating** b **causes** $a \mid true$ **if** $\neg b$
2. **initiating** b **causes** $e \mid true$ **if** $\neg b$
3. **initiating** $a \wedge \neg e$ **causes** $\neg d \mid true$ **if** $\neg a$
4. **initiating** $\neg d$ **causes** $c \mid true$ **if** d

Assume then a state S in which $\neg a, \neg b, \neg c, d$ and $\neg e$ are true, and an action A which initiates b. Using the causal rules, we can obtain a candidate successor state by applying rules 1 and 2 in any order (none of the intermediate states satisfies the state constraints), which leads to the state $\{a, b, \neg c, d, e\}$. No other rules are applicable in that state, but it still violates the state constraint $\neg a \vee c$. A second candidate successor state can be obtained by consecutively applying rules 1 and 3, followed by 2 and 4 in any order. Again, no intermediate state satisfies the constraints, but the final state $\{a, b, c, \neg d, e\}$ does, and therefore is a successor state.

Using the derived effect rules to compute a successor state, we need to apply the first two rules to satisfy the state constraints $\neg b \vee a$ and $\neg b \vee e$, and

the other rules are not applicable as a result. Hence, we obtain Thielscher's first candidate successor state $\{a, b, \neg c, d, e\}$, which violates $\neg a \vee c$ so is rejected. No other candidate successor state satisfies both $\neg b \vee a$ and $\neg b \vee e$, so we obtain no valid successor state: the action is impossible under the given circumstances.

In summary, we find that the method we have developed only generates effect rules that restore the constraints they are derived from with a minimal set of changes, whereas Thielscher's method also allows for non-minimal changes. It is interesting to note that in the setting proposed by Thielscher, i.e. if causal rules are derived from state constraints with the explicit task of restoring the validity of these constraints, minimal change solutions (adhering to the strong correctness criterion) are to be preferred. This remains the case even when this approach sometimes prohibits a particular action rather than generating a set of ramifications which "by accident" restores some constraint's validity: after all, we have to keep in mind that our goal is not generating ramifications restoring the state constraints at all costs. Rather we want to derive exactly those ramifications that are justified by the domain knowledge. Unjustified ramifications are to be avoided, as inertia is still the most fundamental law governing temporal domains.

On the other hand, if the rules are seen as effect propagation rules only roughly related to state constraints, the non-minimal solutions offered by Thielscher's method need not be rejected. In general we should simply not assume that the state constraints and influence information provide sufficient information to determine the precise intended effect rules, so neither should we impose a strong preference for minimal changes with respect to these state constraints. In our approach this issue is dealt with by an indication of possible underspecification (i.e. the presence of a bad grounding in the semantics of the effect rules), in addition to the generation of rules prescribing minimal changes.

One way to deal with the problem of underspecification could be making influence information more fine-grained. For example, one can imagine saying that a can influence b or c, but not both, or only in certain circumstances. However, if influence information needs to get this detailed, there is little or no reason for not immediately writing the intended effect rules instead.

Due to the fact that Thielscher was the first and up to now only researcher to use influence information, we have extensively compared our approach to his, even though our viewpoints on ramifications show some fundamental differences. In the next section, we address one of the most fundamental of these differences: the different views on delays in change propagations.

7.9 Delayed Causation

Effect propagations, especially in physical systems, usually incorporate very small delays. For all practical purposes these delays can usually be abstracted away and the effects assumed to be simultaneous and instantaneous. This abstraction yields a significant simplification and is adopted in most temporal reasoning approaches, including the one presented here.

Evidently, this abstraction is no longer valid if the presence of delays has macroscopic effects. In this case they must be represented explicitly. The approach of Thielscher we have discussed above is a kind of mixed approach in this respect: on the one hand, delays are abstracted away in the sense that the successor states of each state are obtained by applying a complete batch of effects to this starting state, and nothing can interrupt this batch of effects. On the other hand, the delays are assumed to really exist and to have possible macroscopic effects. In particular contradictory effects, like initiating and terminating a particular fluent in the same batch of effects, are allowed in Thielscher's approach, and between these two changes the momentary value of the fluent may have effects that remain visible, like in Thielscher's light detector example which we will discuss later.

We prefer a different approach for two reasons: first of all, the assumption that ramifications incorporate a small delay is not always valid. This assumption implies for example that all state constraints which are restored by ramifications, are violated for very small periods of time. While for some state constraints this is acceptable, in other cases it leads to counterintuitive results. As an example, assume we have fluents on and off representing the states of a switch. Assume $on \leftrightarrow \neg off$ is a state constraint, then we expect that rules stating that any effect is caused by the switch being on and off at the same time, would never have any effect. However if we assume that the state constraints are violated for small periods of time, these counterintuitive effects can occur any time the switch is toggled.

A second problem is in our view that if there are actual delays that are so pronounced as to have macroscopic effects, the assumption that they can nevertheless be abstracted away is not necessarily valid. For example, it is not clear why no other actions would be allowed to occur during these delays.

For these reasons, we adopt the following approach: usually we assume ramifications to be instantaneous, either because they really are, or because the delays involved are entirely irrelevant. Of course then we assume real instantaneity, and disallow contradictory effects or effects generated by hypothetical intermediate states. This has been our approach up to now.

On the other hand, if delays are relevant and have macroscopic effects,

we model them explicitly in a theory of delayed causation. There are plenty of options for such a theory, which we study in this section.

The basic idea is that an action or event a at a certain time t may cause another action or event b at a later time $t + d$. Possibly this depends on some conditions F at the time of a. This could be represented as

$$a \textbf{ dcauses } b \textbf{ after } d \textbf{ if } F.$$

Another option is to let a particular combination of fluents be the trigger of a later event, as in

$$\textbf{initiating } F' \textbf{ dcauses } b \textbf{ after } d \textbf{ if } F.$$

with F' a fluent formula. Similarly, the delayed effect may not be an event but a fluent change:

$$a \textbf{ ecauses } l \textbf{ after } d \textbf{ if } F.$$

or

$$\textbf{initiating } F' \textbf{ ecauses } l \textbf{ after } d \textbf{ if } F.$$

with l a fluent literal (which in turn might be extended to a disjunction of conjunctions to represent nondeterminism).

Another issue is that a delayed effect may be cancelled if some conditions are changed before it actually occurs. To represent this, we would need to distinguish conditions that need to hold at the time of initial "causation" of the delayed effect and conditions that need to persist until it takes place. For example:

$$a \textbf{ if } F \textbf{ dcauses } b \textbf{ if } F' \textbf{ persists after } d.$$

In all the above cases, it is some action or the initiation of some fluent formula which causes the delayed effect. Related to this but slightly different is the issue of natural events, discussed for example in [90], where events may be (immediately) triggered as soon as some fluent formula holds (rather than as soon as it is initiated). Clearly in the language $\mathcal{ER}$ presented here, this would not make sense as formulae hold only immediately after their initiation, all changes are discrete and in the real numbers there is no time point immediately after another one. The importance of natural events is of course strongly linked to continuous change. In fact, autotermination events and similar events triggered by a continuous fluent, which we have discussed in Chapter 6, are typical examples of natural events. As continuous change is outside the scope of this chapter, we do not discuss natural

events further here, though a treatment of them would be very similar to that of delayed effects.

Whatever the exact form of delayed effect rules, it seems natural to us to read them as a definition of the predicate **Happens**, or as part of the inductive definition of **Causes**. In the former case, it might also be wise to distinguish between "primitive" events (actions performed by some agent) and "delayed effect" events, if only for the reason that the former may be arbitrary while the latter are uniquely determined by what preceded them.

It is not easy to keep the syntax of delayed effect theories simple while having the needed expressive power. The rules are quite complex, requiring many parameters representing different conditions. One thing we can do in the interest of simplicity is restricting the syntax to only one type of rule. To achieve this we need to decide whether the rule body will be an event or an initiation and whether its head will be an event or an initiation. Since in $\mathcal{ER}$ up to now initiations follow from actions at the same time point, the most flexible approach is to have initiations be the cause of later events: the other three cases can then be dealt with by combining immediate *event* $\rightarrow$ *initiation* propagations with delayed *initiation* $\rightarrow$ *event* propagations. Hence we propose rules of the form

initiating F **if** F' **ecauses** e **if** F'' **persists after** d

with the intended reading that if at any time point t, F is strongly initiated while F' holds, and if F'' remains true throughout $]t, t+d]$, then event e occurs at time $t+d$.

As far as the formal semantics is concerned, the idea is to read these rules as part of a (definite) inductive definition on the predicate **Happens**. These rules describe the "caused" events, and an additional set of rules which we will introduce further on defines the occurrence in terms of **Happens** of primitive events.

Note that in the definition of **Happens** no cycles can occur: a delayed event occurrence is uniquely determined by what happened before and what holds at the time of occurrence t, which can be evaluated without referring to any occurrences at t or later time points.

We now define the proposed extension, which we will call $\mathcal{ERD}$, more precisely. First of all, we introduce a new sort $\mathcal{PA}$ of primitive actions, which is a subset of $\mathcal{A}$, and a set $\mathbf{PA} \subseteq \mathbf{A}$ of constants of sort $\mathcal{PA}$. We replace the predicate $\mathbf{Happens} : \mathcal{A} \times \mathcal{T} \rightarrow \mathcal{P}$ by $\mathbf{P_Happens} : \mathcal{PA} \times \mathcal{T} \rightarrow \mathcal{P}$.

Definition 7.9.1 ($\mathcal{ERD}$-**signature**)
An $\mathcal{ERD}$-signature Σ is a tuple $<$**Sorts, Functors, Vars**$>$ *with*

- **Sorts** $= \{\mathcal{T}, \mathcal{A}, \mathcal{PA}, \mathcal{F}, \mathcal{P}\}$, *representing the sorts time, action, primitive action, fluent and atom.*
- **Functors** *consists of*
  - *a set* **T** *of constants of sort* $\mathcal{T}$, *denoted* $t, t_1, \ldots,$ *which includes all real numbers;*
  - *a set* **A** *of constants of sort* $\mathcal{A}$, *denoted* $a, a_1, \ldots;$ **PA** *is a subset of* **A** *of constants of sort* $\mathcal{PA}$
  - *a set* **F** *of constants of sort* $\mathcal{F}$, *denoted* $f, f_1, \ldots;$
  - *four typed predicate symbols* $\leq : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{P}$; **P_Happens** : $\mathcal{PA} \times \mathcal{T} \rightarrow \mathcal{P}$; **Initially** : $\mathcal{F} \rightarrow \mathcal{P}$; **Holds** : $\mathcal{F} \times \mathcal{T} \rightarrow \mathcal{P}$.
- **Vars** = $\mathbf{Vars}_{\mathcal{A}} \cup \mathbf{Vars}_{\mathcal{T}}$, *disjoint infinite sets of variables of sort* $\mathcal{A}$ *resp.* $\mathcal{T}$, *denoted as* $A, A_1, \ldots$ *resp.* $T, T_1, \ldots$.

Definition 7.9.2 ($\mathcal{ERD}$**-formulae)**
Given Σ, *the formulae of* $\mathcal{ERD}$ *are:*

- **direct effect rules** *of the form*

$$a \textbf{ causes } D \textbf{ if } F$$

- **derived effect rules** *of the form*

$$\textbf{initiating } F \textbf{ causes } D \textbf{ if } F'$$

- **delayed effect rules** *of the form*

$$\textbf{initiating } F \textbf{ if } F' \textbf{ dcauses } a \textbf{ if } F'' \textbf{ persists after } d$$

- *any sentence (i.e. formula without free variables) constructed in the usual way of* **Holds, P_Happens,** $\leq$, **Initially** *atoms and the connectives and quantifiers* $\neg, \wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow, \forall, \exists$.

where d *is a positive real number* $(0 < d)$, a *an action,* D *a disjunction of conjunctions of fluent literals, and* F, F', F'' *general fluent formulae.*

Definition 7.9.3 ($\mathcal{ERD}$**-theory)**
An $\mathcal{ERD}$*-theory is a tuple* $< \Sigma, \Pi_e, \Pi_d, \Pi_p >$ *such that* Σ *is an* $\mathcal{ERD}$*-signature,* Π_e *is a set of direct or derived effect rules based on* Σ, Π_d *is a set of delayed effect rules based on* Σ, Π_p *is a set of sentences based on* Σ.

The semantics of $\mathcal{ERD}$ can be defined as follows.

Definition 7.9.4 **($\mathcal{ERD}$-interpretation)**
Given an $\mathcal{ERD}$-signature Σ, a temporal ($\mathcal{ERD}$-)interpretation of Σ is a structure $I = <\mathbf{P}, \mathcal{F}un, \mathcal{H}>$ with:

$$\begin{aligned}
\mathbf{P} = &\{t_1 \le t_2 | t_1, t_2 \in \mathbf{T}\} \cup \\
&\{\mathbf{Initially}(l) | l \in \widehat{\mathbf{F}}\} \cup \\
&\{\mathbf{Happens}(a, t) | a \in \mathbf{A}, t \in \mathbf{T}\} \cup \\
&\{\mathbf{P_Happens}(pa, t) | pa \in \mathbf{PA}, t \in \mathbf{T}\} \cup \\
&\{\mathbf{Holds}(l, t) | l \in \widehat{\mathbf{F}}, t \in \mathbf{T}\} \cup \\
&\{\mathbf{Init}(t, l) | t \in \mathbf{T}, l \in \widehat{\mathbf{F}}\} \cup \\
&\{\mathbf{Causes}(t, l) | t \in \mathbf{T}, l \in \widehat{\mathbf{F}}\}
\end{aligned}$$

$\mathcal{F}un : \mathbf{T} \to \mathbf{R}$, *a mapping of time constants to reals such that each real number is mapped to itself*

$\mathcal{H} : \mathbf{P} \to \{\mathbf{t}, \mathbf{f}\}$, *a truth assignment function.*

$\mathcal{H}$ defines relations interpreting **Happens**, **P_Happens**, **Holds**, **Initially**, $\le$, **Causes** and **Init**; we denote them $\mathcal{H}a, \mathcal{PH}a, \mathcal{H}o, \mathcal{I}nitially, \preceq, \mathcal{C}auses$ and $\mathcal{I}nit$ respectively. An $\mathcal{ERD}$-interpretation needs to satisfy the same general conditions as an $\mathcal{ER}$-interpretation (see section 7.3.3). An $\mathcal{ERD}$-interpretation I is a model of an $\mathcal{ERD}$-theory $<\Sigma, \Pi_e, \Pi_d, \Pi_p>$ iff it is a model of Π_e, Π_d and Π_p. Whether I is a model of Π_p and Π_e or not is defined like in $\mathcal{ER}$, extended with nondeterministic rules as in the previous section. The only thing left to be defined is when I is a model of Π_d. This is done as follows:

Definition 7.9.5 **(grounding of Π_d)**
The grounding of a delayed effect rule

$$\textbf{initiating } F \textbf{ if } F' \textbf{ dcauses } e \textbf{ if } F'' \textbf{ persists after } d$$

is the set:

$$\begin{aligned}
\{\mathbf{Happens}(e, t) \leftarrow\ &\mathcal{H}o(F', \mathcal{F}un(t) - d), \mathcal{I}nit(\mathcal{F}un(t) - d, F), \\
&\mathcal{P}ersists(F'', \mathcal{F}un(t) - d, \mathcal{F}un(t)) \mid t \in \mathbf{T}\}.
\end{aligned}$$

where $\mathcal{H}o(F, t)$ is defined as before, $\mathcal{I}nit(t, F)$ is the truth value of

$$\neg\mathcal{H}o(F, t) \wedge \bigvee_{L_i \cup L_p = supp.\ set\ of\ F} (\mathcal{I}nit(t, L_i) \wedge \overline{\mathcal{I}nit(t, \overline{L_p})} \wedge \mathcal{H}o(L_p, t))$$

and $\mathcal{P}ersists(F, t', t)$ is the truth value of

$$\forall T' : (t' < T') \wedge (T' \preceq t) \to Ho(F, T')$$

The grounding $\mathcal{D}_{delay}$ of Π_d is the union of the groundings of all rules of Π_d.

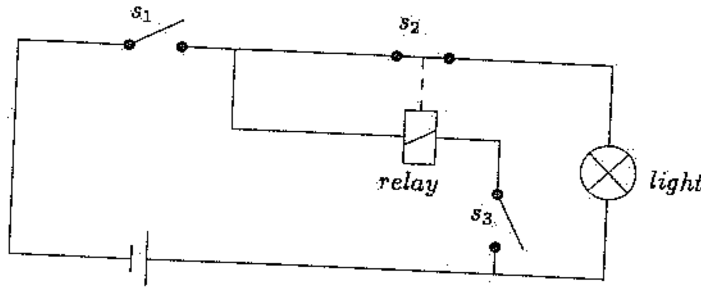

Figure 7.1: Schema of the relay example

Finally, we combine delayed effect events with primitive actions in one definition, as follows:

Definition 7.9.6 (effect definition)
The effect definition $\mathcal{D}_{event}$ of Π_d is
$\mathcal{D}_{delay} \cup \{\mathbf{Happens}(pa,t) \leftarrow \mathbf{P_Happens}(pa,t) \mid t \in \mathbf{T}, pa \in \mathbf{PA}\}$.

$\mathcal{D}_{event}$ is a definite inductive definition on the atom domain $\mathbf{A}' = \{\mathbf{Happens}(a,t) | t \in \mathbf{T}, a \in \mathbf{A}\}$, for which $I_{\mathcal{D}_{event}}$ is defined as $\mathcal{PI}_{\mathcal{D}_{event}}\uparrow$.

We can then complete the definition of model of an $\mathcal{ERD}$-theory:

Definition 7.9.7 ($\mathcal{ERD}$-model)
Given an $\mathcal{ER}$-theory $\Pi_{\mathcal{ER}} = <\Sigma, \Pi_e, \Pi_d, \Pi_p>$, a temporal interpretation I is a model of $\Pi_{\mathcal{ER}}$, denoted $I \models \Pi_{\mathcal{ER}}$, iff $I \models \Pi_p$, $I \models \Pi_d$ and $I \models \Pi_e$, where
$I \models \Pi_p$ *iff* $\forall F \in \Pi_p : \mathcal{H}(F) = \mathbf{t}$.
$I \models \Pi_d$ *iff* $\forall t \in \mathbf{T}, a \in \mathbf{A} : \mathcal{H}a(a,t) \leftrightarrow I_{\mathcal{D}_{event}}(\mathbf{Happens}(a,t))$.
$I \models \Pi_e$ *iff* $\forall t \in \mathbf{T}, l \in \widehat{\mathbf{F}}$:
$\mathcal{I}nit(t,l) \leftrightarrow I_{\mathcal{D}_{init}}(\mathbf{Init}(t,l))$ *and* $Causes(t,l) \leftrightarrow I_{\mathcal{D}_{init}}(\mathbf{Causes}(t,l))$.

As an example, we present the relay example from [109], which is in that paper represented without explicit delays as an example of Thielscher's view on and approach to indirect effects. As we indicated before, in our approach this example should be modelled with explicit delays. The example is an electric circuit with a light, several switches and a relay, as shown in Figure 7.1.

One switch s_1 is serially connected to a system of two parallel wires. On one of these wires we find a switch s_2 and a light, on the other we find a switch s_3 and a relay which operates s_2. The state constraints

are $(s_1 \wedge s_2) \leftrightarrow light$ and $(s_1 \wedge s_3) \leftrightarrow relay$, with influence information $\{(s_1, light), (s_2, light), (s_1, relay), (s_3, relay)\}$, leading to a set of derived effect rules

initiating $s_1 \wedge s_2$ **causes** $light \mid true$ **if** $true$
initiating $\neg s_1 \vee \neg s_2$ **causes** $\neg light$ **if** $true$
initiating $s_1 \wedge s_3$ **causes** $relay \mid true$ **if** $true$
initiating $\neg s_1 \vee \neg s_3$ **causes** $\neg relay \mid true$ **if** $true$

In addition, we need to model the effect of the relay. The relay operates s_2, and we assume there is some delay involved between the activation of the relay and its effect. So we write (omitting the "if true" conditions and assuming the state of the relay needs to persist until the effect occurs):

initiating $relay$ **dcauses** $\neg s_2$ **if** $relay$ **persists after** d
initiating $\neg relay$ **dcauses** s_2 **if** $\neg relay$ **persists after** d

Then, in case s_2 and s_3 are closed (i.e. true) but s_1 is open (false), the closing of s_1 will have the effect of turning on the light and activating the relay. Then, after a time period of length d and unless other actions occur which influence the relevant fluents, the relay will cause s_2 to open and the light to dim again.

Thielscher has extended this example to incorporate a detector, assumed to turn (and then remain) on as soon as light shines on it. We can model the behaviour of this detector by the rule

initiating $light$ **dcauses** $detect$ **if** $light$ **persists after** d'

assuming that there is a threshold duration d' for light to be detected. Thielscher uses this example to illustrate nondeterminism arising from the causal rules. Indeed, whether or not the detector will detect light in the above situation, depends on which of d or d' is greater, which is left implicit in Thielscher's approach. If we assume d and d' to be unknown positive time constants, we also obtain nondeterminism. On the other hand, if d and d' are known (which we assume), the outcome of closing s_1 is uniquely determined.

7.9.1 Mapping Delayed Effect Rules to OLP

Finally, we need to map delayed effect rules to OLP. This can be done as follows. First of all, as the introduction of delayed effect rules required a modification to the syntax of $\mathcal{ER}$ (introducing a new predicate **P_Happens** which replaces **Happens** in Π_p), we need to modify the mapping accordingly. This is done by treating **P_Happens** exactly as we treated **Happens**

before, i.e. by mapping each atom $\mathbf{P_Happens}(\alpha, \tau)$ in Π_p formulae to $phappens(\alpha, \tau)$ in the corresponding FOL axiom and declaring *phappens* an open predicate. *happens* now no longer occurs in FOL axioms (since **Happens** does not occur in Π_p). It is a defined predicate of which the definition follows below. To distinguish between primitive and non-primitive actions, we simply introduce a predicate *primitive*/1 defined by enumeration. Moreover we add a FOL axiom $phappens(A, T) \to primitive(A)$ to the theory.

Then, recall that the effect definition $\mathcal{D}_{event}$ of a set of delayed effect rules Π_d is $\mathcal{D}_{delay} \cup \{\mathbf{Happens}(pa, t) \leftarrow \mathbf{P_Happens}(pa, t) \mid t \in \mathbf{T}, pa \in \mathbf{PA}\}$. Hence, the mapping of Π_d to OLP is a definition of *happens* which consists of some clauses for each delayed effect rule, plus one general clause. This general clause is simply

$$happens(A, T) \leftarrow phappens(A, T).$$

The other clauses are obtained as follows: a rule

$$\textbf{initiating } F \textbf{ if } F' \textbf{ dcauses } a \textbf{ if } F'' \textbf{ persists after } d$$

is mapped to

$$\begin{array}{lll} \{happens(a, T) & \leftarrow & T = T' + d,\ \bigwedge_{l \in L_1} causes(T', l), \\ & & \bigwedge_{l \in L_2} (holds(l, T'), \neg causes(T', \bar{l})), \\ & & \neg holds(F, T'), holds(F', T'), persists(T', F'', T). \\ & | & L_1 \cup L_2 \text{ is a supporting set of } F\} \end{array}$$

where *persists* is defined as

$$\begin{array}{lll} persists(T', F, T) & \leftarrow & holds(F, T'), \neg clipped(T', F, T). \\ persists(T', F, T) & \leftarrow & causes(T', F), \neg clipped(T', F, T). \end{array}$$

and *clipped* as defined before.

The correctness proof of this mapping can largely be based on the proof of the case without delays in section 7.5. As before, it is clear that Π_p and the FOL axioms in OLP are equivalent. Following the same reasoning as in section 7.5, we find that it suffices to prove that, given *initially* for fluent atoms, *phappens* and $\leq$, and their exact counterparts in $\mathcal{ER}$, we find the same truth value for all other atoms in both formalisms. For complex *initially* and $\mathcal{I}nitially$ atoms this is trivial since we use the same inductive definition in terms of simple atoms in both formalisms. This leaves us with simple and complex *holds* atoms, *causes* atoms and *happens* atoms.

We can again use induction on events. First of all, note that if the set of primitive events satisfies the well-founded topology requirement, then so

does the set of all time points that possibly give rise to an event, provided that all delays are finite positive constants and that there is a finite number of delayed effect rules. This set of possible events contains all t' of the form $t + \sum_i k_i d_i$, where t is a primitive event, the d_i are the delay constants occurring in delayed effect rules and the k_i are natural numbers. No event can occur at any other time.

That this set has a well-founded topology follows from the following observations. First, there is a first element, which is the first primitive event as there are no negative delays. Second, the number of possible events in a finite time interval is finite, which we prove by induction on time intervals of length d_{min}, with d_{min} the minimal delay constant in the theory, as follows. There is only one event in $[e_{start}, e_{start} + d_{min}[$, and if there is a finite number of events in $[e_{start}, t[$ then there is only a finite number of events in $[t, t + d_{min}[$. The latter statement follows from the fact that each non-primitive event in $[t, t + d_{min}[$ must be caused by an event in $[e_{start}, t[$, which is a finite set in which each event can only cause a finite number of new events. In addition, $[t, t + d_{min}[$ may contain primitive events, but since these form a well-founded topology their number is also finite. Hence, the set of all time points that are possible events is well-founded.

Using this well-founded topology we prove the equivalence of $holds(f,t)$ with $\mathcal{H}o(f,t)$, of $holds(F,t)$ with $\mathcal{H}o(F,t)$, of $happens(a,t)$ with $\mathcal{H}a(a,t)$, and of $causes(t,l)$ with $Causes(t,l)$ by induction on possible events. The equivalence of $holds(F,t)$ with $\mathcal{H}o(F,t)$ for time points before the first event is proven exactly like in section 7.5. We then prove consecutively for each possible event e that $happens(a,e) \leftrightarrow \mathcal{H}a(a,e)$, that $causes(e,l) \leftrightarrow Causes(e,l)$ and that for all time points t between e and the next possible event $holds(f,t) \leftrightarrow \mathcal{H}o(f,t)$ and $holds(F,t) \leftrightarrow \mathcal{H}o(F,t)$.

If we can prove the first step for a particular event, the proof of the other three steps is exactly the same as in section 7.5, so we only need to prove that $happens(a,e) \leftrightarrow \mathcal{H}a(a,e)$ if all predicates in OLP and $\mathcal{ER}$ coincide for all $t < e$.

This result is obtained as follows. It follows from the inertia axiom and the initiation consistency condition that

$$[\mathcal{H}o(F,T) \vee Causes(T,F)] \wedge \neg\exists T' : [Causes(T'', \neg F) \wedge T \leq T' \wedge T' < e]$$

is equivalent to

$$\forall T' : [(T < T' \wedge T' \leq e) \rightarrow \mathcal{H}o(F,T')]$$

Since all predicates in OLP and $\mathcal{ER}$ coincide for all time points before e, the former formula is equivalent to $persists(T,F,e)$, while the truth value of the latter is $\mathcal{P}ersists(T,F,e)$. Hence, the truth value of $persists(T,F,e)$ is

$Persists(T, F, e)$ for all F and all $T < e$. In addition, the entire definitions of *happens* and **Happens** only refer to time points before e, so it follows that they are equivalent. The correctness of the mapping is thereby proven.

7.10 Related work

We have already compared our work in detail with the approach in [109], which is most similar to ours. For this reason we can refer to [109] for a comparison with approaches not based on causal laws (e.g. categorisation based approaches like [63], [64], [14]): with respect to those approaches $\mathcal{ER}$ and Thielscher's proposal have the same advantages. The most important differences between Thielscher's approach and ours are that Thielscher's causal rules are strongly coupled with (derived from and used in combination with) state constraints, and that Thielscher abstracts away all delays at a macroscopic level but retains them at a microscopic level, whereas we either abstract delays away entirely or represent them explicitly. Due to the former difference the full effect of syntactically uncoupling causal laws from state constraints (which is only achieved in Thielscher's approach and ours) is partially lost. For example, no state constraint independent effect propagations can be represented. Nevertheless, the fact that influence information is represented independent of the state constraints makes Thielscher's approach very appropriate for analysing other proposals using causal laws, which we will do.

The approach to ramifications in the $\mathcal{E}$ language ([50]) can be interpreted as a more coarse-grained variant of Thielscher's: it uses formulae *A whenever C*, with A a fluent and C a set of fluent literals to be read as a conjunction. Such a formula corresponds to a combination of the state constraint $A \leftarrow \bigwedge_{c \in C} c$ with influence information stating that each fluent in C influences the fluent A. As a result of this tight coupling of influence information and state constraint, it is not possible to represent some of the more fine-grained influences that can be represented in Thielscher's approach. However, it should be noted that dealing with ramifications was not a major goal of $\mathcal{E}$. In other respects, the $\mathcal{E}$ language is closer to $\mathcal{ER}$ than Thielscher's approach, in particular in its use of an event-based time structure modelled after the Event Calculus. Apart from less stress on ramifications, an interesting point of difference with $\mathcal{ER}$ is also that $\mathcal{E}$ is mapped to standard logic programming rather than OLP, using an autoepistemic view on LP rather than the definitional view we prefer. One consequence of this is that the mapping of $\mathcal{ER}$ (to OLP) is sound and complete, whereas the mapping of $\mathcal{E}$ (to LP) is sound but not complete.

Returning to the issue of ramifications, we should also consider the

approach in [70], where the need for causal laws is clearly motivated and where causal laws are presented as so-called *S-conditionals*, i.e. formulae $\phi \Rightarrow \psi$ with ϕ and ψ propositional formulae, in an extension of $S5$ modal logic. The reading of such a law is that ϕ determines the truth of ψ: it entails the state constraint $\neg\phi \vee \psi$, plus in case ψ is a literal the influence information that literals in ϕ influence ψ. If ψ is not a literal, the picture gets more complicated: then all literals in ϕ can influence all literals in ψ and all literals in ψ can influence each other. In this respect the proposal is more general than the $\mathcal{E}$ approach. In any case, it is clear that like in $\mathcal{E}$ the causal laws entail the corresponding state constraints, which is the most essential difference with our approach.

Another similar approach is the proposal in [65] based on the situation calculus. Lin introduces a new predicate $caused(p, v, s)$ meaning that proposition p is *caused* to have truth value v in state s. This predicate is circumscribed to minimise change. Ramifications are represented by formulae using the *caused* predicate, e.g. for the suitcase example $up(l_1, s) \wedge up(l_2, s) \rightarrow caused(open, true, s)$ represents that if both latches are open, then the suitcase is caused to be open. The above formula entails the state constraint $up(l_1, s) \wedge up(l_2, s) \rightarrow open(s)$, and incorporates moreover the influence information that l_1 and l_2 may influence *open*. Note that the condition of the rule is a complex formula, making it similar to a complex derived effect rule in $\mathcal{ER}$. However, the causal rules differ from the ones in $\mathcal{ER}$ in that they also entail the corresponding state constraint and in the fact that the minimisation policy does not allow for cyclic dependencies.

The approach in [42] is based on the *Features and Fluents* framework ([96]), and in that sense differs considerably from ours as far as basic concepts are concerned. However, there are some interesting correspondences at a higher level. As an example, the circumscription policy consisting of a combination of minimising and filtering corresponds to our use of inductive definitions for effect rules (minimising changes) and first order logic for observations, action preconditions and state constraints (used to filter interpretations). An important difference with our approach is (apart from the formal details of syntax and semantics) the fact that actions are considered to have duration. Another difference is that time is considered to be isomorphic to the natural numbers. In a sense this corresponds to our condition of well-founded event topology, i.e. *events* in $\mathcal{ER}$ could be mapped to the natural numbers in an order-preserving way, but we consider it more natural to see *time* itself as a full real line. For dealing with ramifications, [42] contains expressions (and formulae incorporating these) of the form $[t]\delta >> [s]\gamma$ where δ and γ are fluent formulae and t and s temporal expressions such that $t \leq s$. This allows for dealing with both immediate (if $t = s$) and delayed ramifications. The formula $[t]\delta >> [s]\gamma$ is defined

to mean $[([t]\delta \rightarrow [s]\gamma) \wedge (([t-1]\neg\delta \wedge [t]\delta) \rightarrow [s]X(\gamma))]$, which basically says that $\mathbf{Holds}(\delta, t) \rightarrow \mathbf{Holds}(\gamma, s)$ and that if δ is strongly initiated at t then γ is allowed to change value at s. This is similar to a state constraint plus the influence information that δ may influence γ, with of course the important generalisation that t and s need not be equal, so that one does not only obtain state constraints but also constraints relating fluents at different time points. Hence, the rules represent not only immediate but also delayed ramifications. This is a considerable advantage of [42]'s approach with respect to nearly all other recent proposals.[35] As in the previously discussed approaches, however, we find that the formulae $[t]\delta >> [s]\gamma$ always entail the corresponding general constraint $\mathbf{Holds}(\delta, t) \rightarrow \mathbf{Holds}(\gamma, s)$, and hence in the case of immediate ramifications ($t = s$) they also entail the corresponding state constraint. Change propagation unrelated to a state or general constraint is also in this approach excluded, unlike in $\mathcal{ER}$. Finally, a disadvantage of the approach in [42] with respect to $\mathcal{ER}$ is that cyclic dependencies in causal laws are not dealt with correctly, as indicated by the authors.

In [97] it is argued that approaches to the ramification problem should be able to deal with so-called *downstream* indirect effects. Sandewall gives the example of a lamp connected to two parallel switches, such that closing either switch turns on the lamp. He argues that one should be able to specify the main effect of an action (for example of turning on the lamp) without specifying the operational details of how this is accomplished (by closing either of the switches). An approach to the ramification problem should then be able to use this description and derive the direct effects of the action from the indirect effect that the lamp is turned on. It is argued that this is a problem for causality-based approaches like ours and most of the above ones.

As stated, the issue is indeed problematic. Clearly we cannot write the action law as a direct effect rule, since it would then imply that the lamp is turned on while the switches are untouched. And in $\mathcal{ER}$, direct effect rules are the only constructs representing action laws. However, we argue that the indirect effect should not be explicitly specified by some action law. We agree with Sandewall's argument that it is often interesting to worry only about the main effect of an action and not about how it is achieved, but this issue deserves closer attention. Given Sandewall's problem specification, we see two possibilities: on the one hand, the switches may be considered nothing but operational details. In that case they can be abstracted away altogether, and the turning on of the lamp can be modelled as an action

[35] But note that both the more general constraints between fluents at different time points and explicit delayed ramification rules can be represented in $\mathcal{ER}$.

with the plain direct effect that the lamp is on. On the other hand, it may occur that the position of the switches is relevant in the domain, but that in a particular application one is only interested in the state of the lamp. This can happen when one gives an agent the task to turn on the lamp, or when turning on the lamp is a necessary step in a plan. But in that case, one can simply model the domain using the usual direct and derived effect rules, and impose in the application at hand that the lamp should be on after a particular action or after a particular plan (imposing this can be done with a simple FOL axiom). Then the agent can find the primitive actions yielding the intended effect on the lamp (closing either of the switches) simply by abductive planning.[36] This approach adequately tackles the given problem, and we find there is no need to modify the domain representation. In general, we argue that there is no need for the representation to deal with deriving causes from their effects, nor is it even desirable that it should do so: this is a typical (abductive or — if the action law needs to be derived explicitly — inductive) reasoning task and not a representation issue.

7.11 Conclusion

We have presented an event-based language able to deal — in a setting of instantaneous actions and discrete change — with all immediate ramifications and known action qualifications, and with delayed ramifications, possibly in the presence of nondeterministic and simultaneous actions and of incomplete knowledge on action occurrences, action ordering or the initial state of the world. The language allows for change propagations not related to state (or more general) constraints between fluents, and for recursion and cycles in the rules describing change propagation. We have discussed and motivated the types of constructs used in the language to reach all of those goals, and presented a semantics based on first order logic and the principle of inductive definitions. This semantics was chosen due to its closeness to the intuitions underlying in particular effect propagations.

We have mapped the language to OLP Event Calculus and proven the correctness of this mapping. The language provides constructs offering the expressive power of the OLP Event Calculus for representing temporal domains, while restricting the latter formalism in such a way that a correct representation methodology is imposed, avoiding unintended and counter-

[36] An alternative but equivalent view is that the turning on of the lamp is a macro-action (as defined in [32]) which can consist of either of the primitive actions. Deriving a primitive action which satisfies the conditions of the macro-action is also a typical abductive task.

intuitive models.

The presented language has been compared with recent proposals for dealing with ramifications. It is intended to deal with the various problems tackled by previous proposals in one coherent language, and to deal in addition with some unaddressed problems, like cycles in derived effect rules and change propagation unrelated to constraints. The extensions for nondeterministic actions and delayed ramifications deal with some other less basic issues in novel ways. Finally we have illustrated how influence information can be used to help derive some of the constructs in our language (a subset of the derived effect rules) directly from state constraints, adapting and improving the method introduced by Thielscher.

Chapter 8

Conclusion

The goal of this thesis has been to study and apply the possibilities of open logic programming for knowledge representation, in particular in a wide range of domains that change over time, both in the fundamental AI setting and in more immediate applications.

To base our work on a strong theoretical argument we have first established a correspondence with description logics, a class of general knowledge representation languages which currently receives a lot of attention in the AI community. We have shown that OLP partitions information in the same way as a description logic in an A-Box and a T-Box, thus addressing the problem of separating assertional from definitional information. We have shown that the various existing description logics correspond to particular subsets of open logic programming, and that current research on extending description logics gradually brings them closer to the more general open logic programming formalism. From a procedural point of view we have shown that the efficient procedures used for reasoning on description logics are specialised instances of SLDNFA, the procedure we use for reasoning on open logic programs. With these correspondences we have shown the theoretical suitability of open logic programming as a general knowledge representation language.

In the rest of the thesis we have complemented this, focussing on time-dependent domains, by showing on the one hand how OLP Event Calculus deals with fundamental theoretical problems in artificial intelligence, and on the other hand how it applies in more directly application-oriented domains where time plays an important role.

In the AI setting we have first shown the precise correspondence between two widely used formalisms for temporal reasoning, the Situation Calculus

and the Event Calculus. We have indicated the practical implications of the differences between the two formalisms, in particular for counterfactual reasoning, showing that either or both formalisms fall short in some cases. We have proposed a new formalism which generalises both original calculi and deals with all aforementioned counterfactual reasoning cases, including some that are handled by neither of the original calculi. Moreover we have precisely shown which restrictions each of the original calculi imposes on the general formalism. Apart of providing a clear relation between situations in Situation Calculus and time points in Event Calculus, this analysis mainly addresses the relevance of the choice between time topologies (in particular linear vs branching time). The main advantage of branching time is that it allows for counterfactual reasoning within the formalism. When counterfactual reasoning is not or not much of importance, or if it can be left to meta-reasoning, the simpler and more natural linear time structure is to be preferred. For this reason, we have worked with the linear time Event Calculus throughout the rest of the thesis.

Also in the AI setting we have developed a very expressive specialised high-level language for representing temporal domains, in the style of $\mathcal{A}$. We have analysed in detail the constructs needed to deal with the frame problem in a very general setting, and brought them together in a coherent framework. The language deals with inertia, known qualifications, immediate and delayed ramifications caused by changes in simple or complex fluent formulae, nondeterministic actions and ramifications, simultaneous actions, general domain constraints (including state constraints and observations at arbitrary time points), and complete or incomplete scenario information in a linear time setting. The language extracts central constructs from the Event Calculus (sometimes in a slightly modified or extended form) but has a much more restricted syntax enforcing a correct methodology for writing specifications. The language has been designed to have all the useful expressive power of the Event Calculus, without the risks of abuse of too much freedom. The exact relation has been clarified by a mapping of the language to OLP Event Calculus, of which we have proved the correctness. We have compared the language with recent work, in particular with recent approaches to the ramification problem, and shown that it deals with a wider class of ramifications than any of these. Moreover it deals at the same time with all of the issues listed above. Finally we have studied the issue of using influence information to derive part of the domain theory semi-automatically in the context of this language. We have adapted and improved a method introduced by Thielscher for automatically deriving derived effect rules from given state constraints and influence information.

On the more application-oriented side we have presented three contributions in this thesis. A first contribution is an extension of the OLP Event Calculus for dealing with qualitative information on continuous change. We have exploited the power of open logic programming for representing partial knowledge on the domain to provide a qualitative characterisation of changing fluents. By distinguishing between continuous changes and continuous influences on these changes, we were able to model complex behaviour of fluents in a concise way. The axioms representing continuous change in the extension have been written in a form very similar to the frame axioms of the discrete Event Calculus, and a smooth integration of continuous and discrete change has been established. Moreover we have shown how the usual forms of reasoning on Event Calculus specifications extend naturally and without modification to domains incorporating continuous fluents, as should of course be intended.

In a second contribution we have used the OLP Event Calculus as the basis for representing a temporal knowledge base with incomplete information. We have extended the Event Calculus with formulae dealing with intervals, so that both information on time points and on intervals can be represented. The representation of the knowledge base has been split up according to the description logics methodology, in a T-Box defining basic formulae in terms of Event Calculus primitives, and an A-Box containing the actual data. We have shown how the basic functionality of a knowledge base can in principle be provided by SLDNFA, and how the knowledge base can be used in general applications, in particular planning, in the same way as usual Event Calculus theories. We have also indicated how SLDNFA can be used as the basis of tools for handling the knowledge base itself, for example for proposing ways of resolving inconsistency by removing certain data items. For dealing with complex FOL formulae in the knowledge base we have implemented the existing transformation of Lloyd and Topor in a preprocessing step applied before calling SLDNFA. The contribution should be considered a theoretical framework, offering a correct representation of a temporal knowledge base and sound procedures for reasoning on it, plus an easily usable interface to applications. To be useful in practice the procedures require a lot of improvement, but this issue is not addressed in this thesis.

A third contribution has been the use of OLP Event Calculus in the area of protocol specification. We have shown how a typical communication protocol can be specified in OLP Event Calculus. Despite the fact that Event Calculus is a much more general formalism, our specification is of a length comparable to that of specifications in specialised process algebras. The specification style is very different, however: where process algebras specify processes as static algebraic objects representing sets of possible

event sequences, in OLP Event Calculus a process is a dynamic entity of which the internal state (and its change over time), the conditions under which actions are possible and the effects of these actions on the world are modelled. The latter specification is more general and contains a lot more information: the possible sequences of events can be derived from the action effects and preconditions, which are abstracted away in process algebra specifications. As a result an Event Calculus specification has the advantage that it can also be used in other applications, for example network management. A related disadvantage is that using Event Calculus specifications for protocol verification is very inefficient. We have included some proofs of properties of the Event Calculus specification in an appendix, but these have not been generated automatically. Specialised support for such proofs should be added to SLDNFA to keep the complexity under control. To allow for an integration of specifications in classical process algebras and in our formalism, we have indicated how process algebra specifications can be mapped to a form which can be embedded in the Event Calculus, although the then resulting specification of course lacks the advantages of a usual Event Calculus specification.

With the contributions in this thesis we hope to establish open logic programming in combination with Event Calculus as a powerful general knowledge representation framework, which is both theoretically sound (as shown by our contributions in the AI setting) and sufficiently expressive to represent a wide range of practical problem domains (as shown by the more general and application-oriented contributions).

The inefficiency of the SLDNFA procedure, of which only a prototype implementation exists, is at this time still a disadvantage for the practical use of the representations.[1] The efficiency issue falls outside the scope of this thesis, which is concerned with knowledge representation, but new projects will in the near future work on efficient implementations.

[1] Even though the framework has been successfully used in several practical applications by other researchers at the department, in particular the scheduling of power plant maintenances of an electricity company.

Bibliography

[1] P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland Publishing Company, 1977.

[2] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *CACM*, 26(11):832–843, 1983.

[3] J. F. Allen. Towards a General Theory of Action and Time. *Artifical Intelligence*, 23(11):123, 1984.

[4] K. Apt and M. Bezem. Acyclic programs. In *Proc. of the International Conference on Logic Programming*, pages 579–597. MIT press, 1990.

[5] F. Baader and U. Sattler. Number restrictions on complex roles in description logics : a preliminary report. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 328–339. Morgan Kaufman, 1996.

[6] A. Baker. Nonmonotonic Reasoning in the Framework of the Situation Calculus. *Artifical Intelligence*, 49:5–23, 1991.

[7] G. Birkhoff. *Lattice Theory*. American Mathematical Society Colloquium Publications, Vol. 25, 1973.

[8] T. Bolognesi, J. van de Lagemaat, and C. Vissers. *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publishers, Boston, 1984.

[9] R. Brachman. On the epistemological status of semantic networks. In N. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*. Academic Press, 1979.

[10] R. J. Brachman, R. Fikes, and H. Levesque. Krypton: A functional approach to knowledge representation. *IEEE Computer*, 16(10):67–73, 1983.

[11] R. J. Brachman, R. Fikes, and H. Levesque. KRYPTON: Integrating Terminology and Assertion. In *Proceedings of the National Conference on Artificial Intelligence*, pages 31–35. William Kaufman, 1983.

[12] R. J. Brachman, V. Gilbert, and H. Levesque. An essential hybrid reasoning system: Knowledge and symbol level accounts of KRYPTON. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 1985*, pages 532–539, 1985.

[13] R. J. Brachman and H. Levesque. Competence in Knowledge Representation. In *Proceedings of the National Conference on Artificial Intelligence*, pages 189–192, 1982.

[14] G. Brewka and J. Hertzberg. How to do things with worlds: on formalizing actions and plans. *Journal of Logic and Computation*, 3(5):517–532, 1993.

[15] M. Buchheit, F. Donini, W. Nutt, and A. Schaerf. A refined architecture for terminological systems: Terminology = schema + views. In *Proc. of the 12th National Conference on Artificial Intelligence*, 1995.

[16] W. Buchholz, S. Feferman, W. Pohlers, and W. Sieg. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies.* Springer-Verlag, Lecture Notes in Mathematics 897, 1981.

[17] D. Calvanese. Finite model reasoning in description logics. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 292–303. Morgan Kaufman, 1996.

[18] W. Chen and D.S. Warren. C-logic of complex objects. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM New York, 1989.

[19] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and databases*, pages 293–322. Plenum Press, 1978.

[20] L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[21] G. De Giacomo and M. Lenzerini. Tbox and abox reasoning in expressive description logics. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 316–327. Morgan Kaufman, 1996.

[22] J. de Kleer and J. S. Brown. A qualitative physics based on confluences. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World*, pages 109–183. Ablex, 1985.

[23] M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming*. PhD thesis, Department of Computer Science, K.U.Leuven, 1993.

[24] M. Denecker. Inductive Definitions, Logic Programming, Knowledge Representation. Technical report, K.U. Leuven, 1996.

[25] M. Denecker and D. De Schreye. SLDNFA; an abductive procedure for normal abductive programs. In K. Apt, editor, *Proceedings of the International Joint Conference and Symposium on Logic Programming, Washington*, 1992.

[26] M. Denecker and D. De Schreye. Justification semantics: a unifying framework for the semantics of logic programs. In *Proc. of the Logic Programming and Nonmonotonic Reasoning Workshop*, pages 365–379, 1993.

[27] M. Denecker and D. De Schreye. Representing incomplete knowledge in abductive logic programming. *Journal of Logic and Computation*, 5(5):553–578, Sept. 1995.

[28] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proceedings of ECAI 92, Vienna*, 1992.

[29] M. Denecker, K. Van Belleghem, G. Duchatelet, F. Piessens, and D. De Schreye. A realistic experiment in knowledge representation in open event calculus : Protocol specification. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming, 1996*, pages 170–184, 1996.

[30] F. Donini, B. Hollunder, M. Lenzerini, D. Nardi, A. M. Spaccamela, and W. Nutt. The complexity of existential quantification in concept languages. *Artificial Intelligence*, 53:309–327, 1991.

[31] C. Evans. Negation as failure as an approach to the Hanks and McDermott problem. In *Proceedings of the second International Symposium on Artificial Intelligence*, 1989.

[32] C. Evans. The Macro-Event Calculus: Representing Temporal Granularity. In *Proceedings of PRICAI, Tokyo*, 1990.

[33] M. Fitting. *First-order Logic and Automated Theorem Proving*. Springer-Verlag, Texts and Monographs in Computer Science, 1990.

[34] K. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[35] A. Galton. A critical examination of Allen's theory of action and time. *Artificial Intelligence*, 42:109–188, 1990.

[36] A. Galton. An investigation of non-intermingling principles in temporal logic. *Journal of Logic and Computation*, 6:271–294, 1996.

[37] M. Gelfond and V. Lifschitz. Describing Action and Change by Logic Programs. In *Proc. of the 9th Int. Joint Conf. and Symp. on Logic Programming*, 1992.

[38] M. Gelfond, V. Lifschitz, and A. Rabinov. What Are the Limitations of the Situation Calculus. In S. Boyer, editor, *Automated reasoning, Essays in Honor of Woody Bledsoe*, pages 167–181. Kluwer Academic Publishers, 1991.

[39] M. Ghallab, J.-P. Haton, J. Hertzberg, E. Sandewall, and E. Tsang. Project temprer. Technical report, 1991.

[40] M. Ginsberg and D. Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35, 1988.

[41] M. Ginsberg and D. Smith. Reasoning about action II: The qualification problem. *Artificial Intelligence*, 35, 1988.

[42] J. Gustafsson and P. Doherty. Embracing occlusion in specifying the indirect effects of actions. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 87–98. Morgan Kaufman, 1996.

[43] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logic, and the frame problem. In *Proceedings of the National Conference on Artificial Intelligence, Philadelphia*, pages 328–333, 1986.

[44] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.

[45] J. Herbrand. Investigations in proof theory. In J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic*, pages 525–581. Harvard University Press, 1967.

[46] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[47] B. Hollunder. Hybrid inferences in KL-ONE-based knowledge representation systems. In *German National Conference on Artificial Intelligence, 1990*, 1990.

[48] B. Hollunder, W. Nutt, and M. Schmidt-Schauss. Subsumption algorithms for concept description languages. In *Proceedings of the 9th European Conference on Artificial Intelligence, Stockholm, 1990*, 1990.

[49] A. Kakas and P. Mancarella. Generalised stable models: a semantics for abduction. In *Proc. of the European Conference on Artificial Intelligence*, 1990.

[50] A. Kakas and R. Miller. A simple declarative language for describing narratives with actions. *Journal of Logic Programming, Special Issue on Reasoning about Action and Change*, 31(1-3), 1997.

[51] A. Kakas and M.Michael. Integrating Abductive and Constraint Logic Programming. In *Proceedings of the Twelfth International Conference on Logic Programming, ICLP-95*, pages 399–417. MIT press, 1995.

[52] A. C. Kakas, R. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[53] G. Karjoth. Stepwise specification of a sliding-window protocol by means of process algebra. Switzerland Chapter on Digital Communication Systems, 1988.

[54] G. N. Kartha. Soundness and completeness theorems for three formalizations of action. In *Proc. of IJCAI-93*, pages 724–729, 1993.

[55] G. N. Kartha and V. Lifschitz. Actions with indirect effects (preliminary report). In *Proc. of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 341–350, 1994.

[56] R. Kowalski. Predicate logic as a programming language. In *Proc. of IFIP 74*, pages 569–574. North-Holland, 1974.

[57] R. Kowalski and F. Sadri. The situation calculus and event calculus compared. In M. Bruynooghe, editor, *Proceedings of ILPS 1994*, 1994.

[58] R. A. Kowalski. Database updates in the event calculus. *Journal of Logic Programming, 1992*, 1992.

[59] R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(4):319–340, 1986.

[60] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:289–338, 1986.

[61] J.-L. Lassez, V. Nguyen, and E. Sonenberg. Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

[62] V. Lifschitz. Computing Circumscription. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, 1985*, pages 121–127, 1985.

[63] V. Lifschitz. Formal Theories of Action. In F. Brown, editor, *Proceedings of the 1987 Workshop on the Frame Problem in AI*, 1987.

[64] V. Lifschitz. Frames in the Space of Situations. *Artificial Intelligence*, 46:365–376, 1990.

[65] F. Lin. Embracing causality in specifying the indirect effects of actions. In C. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1985–1991, 1995.

[66] F. Lin and R. Reiter. State constraints revisited. *J. of Logic and computation, special issue on actions and processes*, 4:655–678, 1994.

[67] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[68] J. Lloyd and R. Topor. Making PROLOG more expressive. *Journal of logic programming*, 1(3):225–240, 1984.

[69] G. Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6:53–68, 1976.

[70] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In C. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1978–1984, 1995.

[71] J. McCarthy. Circumscription - a form of nonmonotonic reasoning. *Artifical Intelligence*, 13:89–116, 1980.

[72] J. McCarthy and P. 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, 1969.

[73] R. Miller. Situation calculus specifications for event calculus logic programs. In *Proceedings of the Third International Conference on Logic Programming and Non-monotonic Reasoning*, pages 217–230, 1995.

[74] R. Miller and M. Shanahan. Reasoning about discontinuities in the event calculus. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 63–74. Morgan Kaufman, 1996.

[75] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[76] M. Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, New York, 1975.

[77] L. Missiaen. *Localized abductive planning with the event calculus*. PhD thesis, Department of Computer Science, K.U.Leuven, 1991.

[78] L. Missiaen. ECSIM: Discrete event simulation using event calculus. *Journal of Logic and Computation*, 1995.

[79] L. Missiaen, M. Bruynooghe, and M. Denecker. Abductive planning with event calculus. Internal report, Department of Computer Science, K.U.Leuven, 1992.

[80] A. Montanari, E. Maim, E. Ciapessoni, and E. Ratto. Dealing with Time Granularity in the Event Calculus. In *Proceedings of FGCS, Tokyo*, pages 702–712, 1992.

[81] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland Publishing Company, Amsterdam- New York, 1974.

[82] E. Pedanult. Adl: Exploring the middle ground between strips and the situation calculus. In H. L. R. Brachman and R. Reiter, editors, *Proc. of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[83] J. Pinto and R. Reiter. Temporal Reasoning in Logic Programming: A Case for the Situation Calculus. In *Proc. of the International Conference on Logic Programming*, pages 203–221, 1993.

[84] J. A. Pinto. Temporal reasoning in the situation calculus. Technical Report KRR-TR-94-1, Computer Science Dept., University of Toronto, 1994.

[85] A. Porto and C. Ribeiro. Temporal inference with a point-based interval algebra. In *Proceedings of ECAI 92, Vienna*, pages 374–378, 1992.

[86] T. Przymuzinski. Perfect Model Semantics. In *Proceedings of JICSLP 88*, pages 1081–1096, 1988.

[87] R. Quillian. Semantic memory. In M. Minsky, editor, *Semantic Information Processing*. MIT Press, 1968.

[88] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[89] R. Reiter. Formalizing Database Evolution in the Situation Calculus. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 600–609, 1992.

[90] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 2–13. Morgan Kaufman, 1996.

[91] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[92] R.S.Miller and M.P.Shanahan. Narratives in the situation calculus. *Journal of Logic and Computation*, 4(5):513–530, 1994.

[93] F. Sadri and R. Kowalski. Variants of the event calculus. In L. Sterling, editor, *Proc. of the International Conference on Logic Programming, 1995*, 1995.

[94] E. Sandewall. Combining logic and differential equations for describing real-world systems. In *Proceedings 1989 Knowledge Representation Conference*, page 412, 1989.

[95] E. Sandewall. Filter preferential entailment for the logic of action in almost continuous worlds. In *Proceedings of IJCAI 89*, page 894, 1989.

[96] E. Sandewall. *Features and Fluents. A Systematic Approach to the Representation of Knowledge about Dynamical Systems. Volume I.* Oxford University Press, 1994.

[97] E. Sandewall. Assessments of ramification methods that use static domain constraints. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 99–110. Morgan Kaufman, 1996.

[98] L. Schubert. Monotonic solution of the frame problem in the situation calculus. In H.E. Kyburg, Jr et al., editor, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Publishers, 1990.

[99] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of IJCAI 89*, page 1055, 1989.

[100] M. Shanahan. Representing continuous change in the event calculus. In *Proceedings of the 9th ECAI*, page 598, 1990.

[101] M. Shanahan. Towards a calculus for temporal and qualitative reasoning. In *Proceedings of AAAI Symposium, Stanford, 1991*, 1991.

[102] M. Shanahan. A circumscriptive calculus of events. *Artificial Intelligence*, 77:249–284, 1995.

[103] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Commonsense Law of Inertia.* MIT Press, 1997.

[104] Y. Shoham. Nonmonotonic reasoning and causation. *Cognitive Science*, 214:213–252, 1990.

[105] A. S. Tanenbaum. *Computer networks, 2nd ed.* Prentice Hall, 1989.

[106] E. B. T.Bolognesi. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN systems*, 14:25–59, 1987.

[107] M. Thielscher. Representing actions in equational logic programming. In P. Van Hentenrijck, editor, *Proc. of the International Conference on Logic Programming, 1994*, 1994.

[108] M. Thielscher. Causality and the qualification problem. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 51–62. Morgan Kaufman, 1996.

[109] M. Thielscher. Ramification and causality. *Artificial Intelligence*, 89(1-2):317–364, 1997.

[110] K. Van Belleghem, M. Denecker, and D. De Schreye. Representing continuous change in the abductive event calculus. In P. Van Hentenrijck, editor, *Proc. of the International Conference on Logic Programming, 1994*, pages 225–240, 1994.

[111] K. Van Belleghem, M. Denecker, and D. De Schreye. The Abductive Event Calculus as a General Framework for Temporal Databases. In *Proc. of the International Conference on Temporal Logic*, pages 301–316, 1994.

[112] K. Van Belleghem, M. Denecker, and D. De Schreye. Combining situation calculus and event calculus. In L. Sterling, editor, *Proc. of the International Conference on Logic Programming, 1995*, pages 83–98. MIT-press, 1995.

[113] K. Van Belleghem, M. Denecker, and D. De Schreye. On the relation between situation calculus and event calculus. *Journal of Logic Programming, Special Issue on Reasoning about Action and Change*, 31(1-3):3–37, 1997.

[114] K. Van Belleghem, M. Denecker, and D. De Schreye. A strong correspondence between description logics and open logic programming. In L. Naish, editor, *Proc. of the International Conference on Logic Programming, 1997*. MIT-press, 1997.

[115] K. Van Belleghem, M. Denecker, and D. Theseider Dupré. Ramifications in an event-based language. In *Proc. of the Ninth Dutch Artificial Intelligence Conference, 1997*, page to appear, 1997.

[116] A. Van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[117] A. Weigel and R. Bleisinger. Support for resolving Contradictions in Time Interval Networks. In *Proceedings of ECAI 92, Vienna*, pages 379–383, 1992.

Appendix A

The Lloyd-Topor Transformation

In this appendix we present the details of the transformation described in [68]. This transformation maps general logic programs, i.e. sets of general clauses $A \leftarrow W$ with A an atom and W an arbitrary FOL formula, to equivalent normal logic programs. The equivalence is proven in [68].

The transformation proceeds by replacing general clauses by other, simpler general clauses until only normal clauses are left. To this end, the following transformation rules are used.[1]

a) Replace $A \leftarrow W_1, W_2, \ldots, \neg(V \wedge W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, \neg V, \ldots W_n.$
and $A \leftarrow W_1, W_2, \ldots, \neg W, \ldots W_n.$

b) Replace $A \leftarrow W_1, W_2, \ldots, (\forall x_1 \ldots x_m : W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, \neg(\exists x_1 \ldots x_m : \neg W), \ldots W_n.$

c) Replace $A \leftarrow W_1, W_2, \ldots, \neg(\forall x_1 \ldots x_m : W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, \exists x_1 \ldots x_m : \neg W, \ldots W_n.$

d) Replace $A \leftarrow W_1, W_2, \ldots, (V \leftarrow W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, V, \ldots W_n.$
and $A \leftarrow W_1, W_2, \ldots, \neg W, \ldots W_n.$

[1] The rules for "$\leftrightarrow$" and "$\oplus$" do not occur in [68], but their addition is not problematic in any way.

e) Replace $A \leftarrow W_1, W_2, \ldots, \neg(V \leftarrow W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, W, \neg V, \ldots W_n.$

f) Replace $A \leftarrow W_1, W_2, \ldots, (V \vee W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, V, \ldots W_n.$
and $A \leftarrow W_1, W_2, \ldots, W, \ldots W_n.$

g) Replace $A \leftarrow W_1, W_2, \ldots, \neg(V \vee W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, \neg V, \neg W, \ldots W_n.$

h) Replace $A \leftarrow W_1, W_2, \ldots, \neg(\neg W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, W, \ldots W_n.$

i) Replace $A \leftarrow W_1, W_2, \ldots, (\exists x_1 \ldots x_m : W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, W, \ldots W_n.$

j) Replace $A \leftarrow W_1, W_2, \ldots, \neg(\exists x_1 \ldots x_m : W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, \neg p(y_1 \ldots y_k), \ldots W_n.$
and $p(y_1 \ldots y_k) \leftarrow \exists x_1 \ldots x_m : W.$
where p is a new predicate symbol not occurring in the program, and $y_1, \ldots y_k$ the free variables in $(\exists x_1 \ldots x_m : W)$.

k) Replace $A \leftarrow W_1, W_2, \ldots, (V \leftrightarrow W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, V, W, \ldots W_n.$
and $A \leftarrow W_1, W_2, \ldots, \neg V, \neg W, \ldots W_n.$

l) Replace $A \leftarrow W_1, W_2, \ldots, \neg(V \leftrightarrow W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, V, \neg W, \ldots W_n.$
and $A \leftarrow W_1, W_2, \ldots, W, \neg V, \ldots W_n.$

m) Replace $A \leftarrow W_1, W_2, \ldots, (V \oplus W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, V, \neg W, \ldots W_n.$
and $A \leftarrow W_1, W_2, \ldots, W, \neg V, \ldots W_n.$

n) Replace $A \leftarrow W_1, W_2, \ldots, \neg(V \oplus W), \ldots W_n.$
by $A \leftarrow W_1, W_2, \ldots, V, W, \ldots W_n.$
and $A \leftarrow W_1, W_2, \ldots, \neg V, \neg W, \ldots W_n.$

As an example, the general clause

$$p(X) \leftarrow q(X), \forall Y : (r(X,Y) \leftrightarrow r(Y,X)).$$

is rewritten to

$$p(X) \leftarrow q(X), \neg\exists Y : \neg(r(X,Y) \leftrightarrow r(Y,X)).$$

and in subsequent steps to

$$\begin{array}{lcl} p(X) & \leftarrow & q(X), \neg h(X). \\ h(X) & \leftarrow & \exists Y : \neg(r(X,Y) \leftrightarrow r(Y,X)). \end{array}$$

and

$$\begin{array}{lcl} p(X) & \leftarrow & q(X), \neg h(X). \\ h(X) & \leftarrow & \neg(r(X,Y) \leftrightarrow r(Y,X)). \end{array}$$

and finally to

$$\begin{array}{lcl} p(X) & \leftarrow & q(X), \neg h(X). \\ h(X) & \leftarrow & r(X,Y), \neg r(Y,X). \\ h(X) & \leftarrow & \neg r(X,Y), r(Y,X). \end{array}$$

which is a normal logic program. More examples can be found in Chapters 3 and 6.

Appendix B

Proof of Lemma 5.5.3

Lemma 5.5.3 states that, given the theory T defined in section 5.5:

$$T \models$$
$$\forall P, A, S : [situation(S) \,\&\, action(A)] \rightarrow$$
$$[(holds'(P, S) \leftrightarrow holds_in(P, S)) \rightarrow$$
$$(holds'(P, result(A, S)) \leftrightarrow holds_in(P, result(A, S))]$$

Proof:
We need to prove for all A,P and S that, given $situation(S)$ and $action(A)$, and given $holds'(P, S) \leftrightarrow holds_in(P, S)$, it follows that

$$holds'(P, result(A, S)) \leftrightarrow holds_in(P, result(A, S))$$

To this end, we rewrite $holds'(P, result(A, S))$ and $holds_in(P, result(A, S))$ first. $holds'(P, result(A, S))$ is by definition equivalent to

$$[holds'(P, S) \ \wedge \ \neg term_s(A, S, P)] \vee init_s(A, S, P)$$

which is the same as (using the induction hypothesis)

$$[holds_in(P, S) \ \wedge \ \neg term_s(A, S, P)] \vee init_s(A, S, P)$$

and using the definitions of $holds_in$,$term_s$ and $init_s$, this is equivalent to

$$[\forall T' : (member(T', S) \rightarrow holds(P, T')) \ \wedge \ \neg\exists E^*, T^* :$$
$$(member(T^*, S) \ \wedge \ event(E^*, T^*) \ \wedge \ act(E^*, A) \ \wedge \ terminates(E^*, P))]$$
$$\vee$$
$$\exists E'', T'' : (member(T'', S) \ \wedge \ event(E'', T'') \ \wedge \ act(E'', A) \ \wedge$$
$$initiates(E'', P))]$$

We will call this formula $\mathcal{F}$.

On the other hand, $holds_in(P, result(A, S))$ is equivalent to

$$\forall T : member(T, result(A, S)) \rightarrow holds(P, T)$$

This formula will be called $\mathcal{G}$.

We need to prove that $\mathcal{F} \leftrightarrow \mathcal{G}$. First we prove that $\mathcal{F} \rightarrow \mathcal{G}$. We assume $\mathcal{F}$ given and prove that $holds(P, T)$ is true for any T for which $member(T, result(A, S))$ holds. We can write $member(T, result(A, S))$ as

$$\exists E', T' : (event(E', T') \wedge T' < T \wedge act(E', A) \wedge member(T', S) \\ \wedge \neg int_events(T', T))$$

We add the disjunction $\mathcal{F}$ to this formula and apply distributivity (i.e. we add this formula to both disjuncts of $\mathcal{F}$):

$$\begin{gathered}[\forall T' : (member(T', S) \rightarrow holds(P, T')) \wedge \\ \neg\exists E', T' : \\ (member(T', S) \wedge event(E', T') \wedge act(E', A) \wedge terminates(E', P)) \wedge \\ \exists E', T' : \\ (event(E', T') \wedge T' < T \wedge act(E', A) \wedge member(T', S) \\ \wedge \neg int_events(T', T))] \\ \vee \\ [\exists E', T', E'', T'' : (event(E', T') \wedge T' < T \wedge act(E', A) \wedge \\ member(T', S) \wedge \neg int_events(T', T)) \wedge event(E'', T'') \wedge \\ member(T'', S) \wedge act(E'', A) \wedge initiates(E'', P))]\end{gathered}$$

Working on the first disjunct, we find that for the E' and T' of its third conjunct it holds on the one hand that $holds(P, T')$ (due to the first conjunct) and on the other hand that $\neg terminates(E', P)$ (due to the second conjunct). We add these two literals and omit the first two conjuncts. This yields

$$\begin{gathered}[\exists E', T' : (event(E', T') \wedge T' < T \wedge act(E', A) \wedge member(T', S) \wedge \\ \neg int_events(T', T) \wedge \neg terminates(E', P) \wedge holds(P, T'))] \\ \vee \\ [\exists E', T', E'', T'' : (event(E', T') \wedge T' < T \wedge act(E', A) \wedge \\ member(T', S) \wedge \neg int_events(T', T)) \wedge event(E'', T'') \wedge \\ member(T'', S) \wedge act(E'', A) \wedge initiates(E'', P))]\end{gathered}$$

Applying lemma 5.5.1.b to the second disjunct, we find that $E' = E''$ and

$T' = T''$:

$$[\exists E', T' : (event(E', T') \wedge T' < T \wedge act(E', A) \wedge member(T', S) \wedge \neg int_events(T'', T) \wedge \neg terminates(E', P) \wedge holds(P, T''))]$$
$$\vee$$
$$[\exists E', T' : (event(E', T') \wedge T' < T \wedge act(E', A) \wedge member(T', S) \wedge \neg int_events(T', T)) \wedge initiates(E', P))]$$

Then we use the definition of *holds* in the first disjunct, and derive in both disjuncts the formula $\neg clipped(T', P, T)$ from the strictly stronger statement $\neg int_events(T', T)$:

$$[\exists E', T', E'', T'' : (event(E', T') \wedge T' < T \wedge act(E', A) \wedge member(T', S) \wedge \neg clipped(T', P, T) \wedge \neg terminates(E', P) \wedge event(E'', T'') \wedge T'' < T' \wedge initiates(E'', P) \wedge \neg clipped(T'', P, T'))]$$
$$\vee$$
$$[\exists E', T' : (event(E', T') \wedge T' < T \wedge act(E', A) \wedge \neg clipped(T', P, T)) \wedge initiates(E', P))]$$

Now, in the first disjunct, from

$$\neg clipped(T', P, T) \wedge event(E', T') \wedge \neg terminates(E', P) \wedge \neg clipped(T'', P, T')$$

and the time relations, we can derive $\neg clipped(T'', P, T)$, since there is no termination of P between T' and T, none between T'' and T' and none at T'. As a result we find that both disjuncts include all literals of the definition of *holds*, so we immediately obtain

$$holds(P, T)$$

which proves $\mathcal{G}$.

Now the only thing left for us to prove is that $\mathcal{G} \rightarrow \mathcal{F}$. This can be proven as follows: $\mathcal{G}$ is, because of lemma 5.5.2, equivalent to

$$\exists T : member(T, result(A, S)) \wedge holds(P, T)$$

or, using the definitions of *member* and *holds*

$$\exists T, E', T', E'', T'' : (T'' < T \wedge event(E'', T'') \wedge initiates(E'', P) \wedge \neg clipped(T'', P, T) \wedge T' < T \wedge event(E', T') \wedge act(E', A) \wedge member(T', S) \wedge \neg int_events(T', T))$$

Here we see $T'' < T$ and $T' < T$, which implies either $T'' < T'$ or $T'' = T'$ ($T' < T''$ is impossible because of $\neg int_events(T', T)$). So we find

$$\begin{array}{c}[\, \exists T, E', T', E'', T'' : (T'' < T' \ \wedge \ event(E'', T'') \ \wedge \ initiates(E'', P) \ \wedge \\ \neg clipped(T'', P, T) \ \wedge \ T' < T \ \wedge \ event(E', T') \ \wedge \ act(E', A) \ \wedge \\ member(T', S) \ \wedge \ \neg int_events(T', T))] \\ \vee \\ [\, \exists T, E', T' : (event(E', T') \ \wedge \ initiates(E', P) \ \wedge \ \neg clipped(T', P, T) \ \wedge \\ T' < T \ \wedge \ act(E', A) \ \wedge \ member(T', S) \ \wedge \ \neg int_events(T', T))]\end{array}$$

From the first disjunct, we can derive on the one hand

$$\begin{array}{c}\exists T, E', T', E'', T'' : (T'' < T' \ \wedge \ event(E'', T'') \ \wedge \ initiates(E'', P) \ \wedge \\ T' < T \ \wedge \ member(T', S) \ \wedge \\ \neg \exists E^*, T^* : [event(E^*, T^*) \ \wedge \ T'' < T^* \ \wedge \ T^* < T \ \wedge \ terminates(E^*, P)])\end{array}$$

and on the other hand

$$\begin{array}{c}\exists T, E', T', E'', T'' : (T'' < T' \ \wedge \ event(E', T') \ \wedge \ act(E', A) \ \wedge \\ T' < T \ \wedge \ member(T', S) \ \wedge \\ \neg \exists E^*, T^* : [event(E^*, T^*) \ \wedge \ T'' < T^* \ \wedge \ T^* < T \ \wedge \ terminates(E^*, P)])\end{array}$$

The second disjunct can be simplified, so that we obtain from the above disjunction

$$\begin{array}{c}[\, \exists T, E', T', E'', T'' : (T'' < T' \ \wedge \ event(E'', T'') \ \wedge \ initiates(E'', P) \ \wedge \\ T' < T \ \wedge \ member(T', S) \ \wedge \\ \neg \exists E^*, T^* : [event(E^*, T^*) \ \wedge \ T'' < T^* \ \wedge \ T^* < T \ \wedge \ terminates(E^*, P)]) \\ \wedge \\ \exists T, E', T', E'', T'' : (T'' < T' \ \wedge \ event(E', T') \ \wedge \ act(E', A) \ \wedge \\ T' < T \ \wedge \ member(T', S) \ \wedge \\ \neg \exists E^*, T^* : [event(E^*, T^*) \ \wedge \ T'' < T^* \ \wedge \ T^* < T \ \wedge \ terminates(E^*, P)])] \\ \vee \\ \exists E', T' : \\ (event(E', T') \ \wedge \ initiates(E', P) \ \wedge \ act(E', A) \ \wedge \ member(T', S))\end{array}$$

which in its turn implies, using the information $T' < T$ in the first conjunct of the first disjunct, and instantiating E^* and T^* to E' and T' in the second

conjunct of that disjunct, that

$$[\exists T', E'', T'':$$
$$(T'' < T' \wedge event(E'', T'') \wedge initiates(E'', P) \wedge member(T', S)$$
$$\wedge \neg\exists E^*, T^*:$$
$$[event(E^*, T^*) \wedge T'' < T^* \wedge T^* < T' \wedge terminates(E^*, P)])$$
$$\wedge$$
$$\exists E', T':$$
$$(event(E', T') \wedge act(E', A) \wedge member(T', S) \wedge \neg terminates(E', P))]$$
$$\vee$$
$$[\exists E', T':$$
$$(event(E', T') \wedge initiates(E', P) \wedge act(E', A) \wedge member(T', S))]$$

Using in the first disjunct the definition of *holds* (on its first conjunct) and lemma 5.5.1.b (on its second conjunct), we find

$$[\exists T' : (member(T', S) \wedge holds(P, T'))$$
$$\wedge$$
$$\forall E', T' : ((event(E', T') \wedge act(E', A) \wedge member(T', S)) \rightarrow$$
$$\neg terminates(E', P))]$$
$$\vee$$
$$[\exists E', T':$$
$$(event(E', T') \wedge initiates(E', P) \wedge act(E', A) \wedge member(T', S))]$$

Finally, we use lemma 5.5.2 and some rewriting on the first disjunct to obtain

$$[\forall T' : (member(T', S) \rightarrow holds(P, T'))$$
$$\wedge$$
$$\neg\exists E', T':$$
$$(event(E', T') \wedge act(E', A) \wedge member(T', S) \wedge terminates(E', P))]$$
$$\vee$$
$$[\exists E', T':$$
$$(event(E', T') \wedge initiates(E', P) \wedge act(E', A) \wedge member(T', S))]$$

which is exactly the formula $\mathcal{F}$ (with some renamed variables).

This completes the proof that in our theory T, for all A, P and S,

$$situation(S) \wedge action(A) \wedge (holds'(P, S) \leftrightarrow holds_in(P, S))$$

implies

$$holds'(P, result(A, S)) \leftrightarrow holds_in(P, result(A, S))$$

□

Appendix C

Correctness Proof of the Sliding Window Protocol Specification

In this appendix we prove that the specification of the sliding window protocol in Chapter 6 is correct, i.e. that it meets the requirement that frames sent out by the network layer on one side arrive on the other side exactly once and in the correct order, unless no packets arrive there at all (in case of a complete breakdown of the channel). We will prove this by first establishing an invariant relation, i.e. a set of statements that are true at each time point. From this invariant relation we will deduce the desired properties.

We prove the correctness under the assumption that there is an initial event (i.e. no infinite sequence of events into the past) and that there is only a finite number of events in a finite time interval (a well-founded event topology as defined in Chapter 7). Under these conditions we can prove invariant relations by induction on events.

We adopt the following conventions:

- Events are ordered according their corresponding time points.
- Frames are denoted by the name of their *net_send* event. The frames sent by one process are assumed to be ordered like their *net_send* events, whereas frames sent by different processes are considered unrelated.
- A frame F has arrived if the receiving process has been in receiving

mode for F.

We assume the number of slots n in each sending window is at least 2 (which is necessary for the protocol to be correct). The protocol starts at $T = 0$. We assume that virtual frames $-n \ldots -1$ are stored in the slots $0 \ldots n-1$ initially, and we say that these have already arrived on the other side. Hence

$$\begin{array}{l} holds(arrived(F), T) \\ \leftarrow [(\exists T' : T' < T \wedge holds((attribute, mode, receiving(F)), T')) \\ \quad \vee (F \in \{-n \ldots -1\})] \end{array}$$

- The notations $A \leq B \leq C$, $A \leq B < C$, $A < B \leq C$ mean that the number B is circularly between A and C (possibly equal to A or C depending on the use of $\leq$ or $<$), and A is not equal to C except possibly in the case $A \leq B \leq C$ provided that $A = B = C$. Likewise, notations like $A \leq B \leq C \leq D$ are used to denote that both B and C are circularly between A and D, and B is encountered before C when going from A to D.

We prove the correctness in two steps: first we determine and prove an invariant relation which is true at all time points. Using this invariant relation we then prove the property given above.

C.1 An Invariant Relation

C.1.1 Specification

We determine an invariant relation which holds at any time T, for any process P. In what follows, any time- or process-dependent parameter A is assumed to denote the value of that parameter of process P at time T. The notation $peer(A)$ denotes the value of parameter A of the peer process of P. Note that parameters may be nested. Then for example $peer(A)(B)$ denotes the value of function A of the peer process of P, with parameter the value B of P itself.

1. $cnext(fts, xpa) \leftrightarrow \neg net_enabled$

2.

$$\begin{array}{ll} (fts \leq SL1 < SL2 = SL1 + 1 \leq fts - 1) \rightarrow & \\ w(SL1) < w(SL2) < T & (2a) \\ \wedge \neg \exists F : w(SL1) < F < w(SL2) & (2b) \\ \wedge \neg \exists F : w(fts - 1) < F < T & (2c) \end{array}$$

3. $cnext(X, xpa) \rightarrow [\forall F' : (F' \leq w(X) \rightarrow peer(arrived)(F'))]$

4. $\forall F' : F' \leq peer(w)(xpf - 1) \rightarrow arrived(F')$

5. $xpa \leq peer(xpf) \leq fts$

6.

$$
\begin{array}{ll}
on_channel(packet(E, F, NR, ACK), peer(P)) \rightarrow & \\
F = w(NR) \vee peer(arrived)(F) & (6a) \\
\wedge\, peer(xpa - 1) \leq ack \leq xpf - 1 & (6b) \\
\wedge\, [(on_channel(packet(E', F', NR', ACK'), peer(P)) & \\
\quad \wedge E < E') \rightarrow peer(xpa - 1) \leq ACK \leq ACK' \leq xpf - 1] & (6c) \\
\wedge\, [peer(xpf) = fts \rightarrow NR \neq fts] & (6d)
\end{array}
$$

7. $mode = sending(F, NR) \rightarrow NR = fts - 1$

C.1.2 Proof

We prove this invariant by induction on events: by the Event Calculus frame axiom, we know that all fluents retain their value between events, and that the value at each event is the same as the value at all times between that event and the previous one. Moreover simultaneous events are disallowed. Hence, if we prove that the invariant is initially true and that each event type preserves the invariant, we can conclude that the invariant is true at all times. We prove that the invariant is preserved for a particular process both after its own actions and after those of its peer process.

In the proof, we use the notation $[i]$ to denote part i of the invariant valid in the previous time interval. E always denotes the event under discussion. We call *oldfts*, *oldxpf*, *oldxpa* resp. *oldw* the values of fts, xpf, xpa and w at the time of (and in the interval before) the event. Observe that attributes always have exactly one value, since they are all initialised and an attribute value can only be terminated through the destructive assignment rule, which is always applicable when a different value is initiated. This unique value property is used implicitly throughout the proof.

Initial state

We first prove that the invariant is true after initialisation:

1. follows from $fts = xpa = 0$ and *net_enabled*

2. a) follows from $w(0) = -n < \ldots < -1 = w(n-1) < 0 < T$
   b) and c) are trivially true as there are no other frames

3. from $cnext(X, xpa)$ it follows that $X = n-1$ and $\forall F' : F' \leq w(n-1) = -1 \rightarrow peer(arrived)(F')$ is true because of the initial assumption

4. $\forall F' : F' \leq peer(w)(n-1) \leftrightarrow arrived(F')$ is true because of the initial assumption for the peer process.

5. $0 \leq 0 \leq 0$ is true

6. *on_channel* is false for all packets

7. $mode \neq sending$

Next, we prove that the invariant is preserved after an event of each type:

Net_send event

$cnext(oldfts, xpa)$ is false ($\Leftarrow$ [1])

1. follows from the fact that *net_enabled* is terminated if and only if $cnext(fts, xpa)$ is initiated

2. already holds for $fts \leq SL1 < SL2 = SL1 + 1 \leq fts - 2$ ($\Leftarrow$ [2])
   a) $w(fts-2) < w(fts-1) = E < T$
   b) $\neg\exists F : w(fts-2) < F < w(fts-1) = E$ ($\Leftarrow$ [2c])
   c) $\neg\exists F : w(fts-1) < F < T$ because no other events happened

3. as xpa does not change, neither does X; w(X) does not change because the only change in w is in $w(oldfts)$ while $oldfts \neq X$ because of $cnext(X, XPA)$ and [1]; therefore the invariant follows from [3]

4. follows from [4]

5. $xpa \leq peer(xpf) \leq fts - 1$ and $fts \neq xpa$ ($\Leftarrow$ [1]), hence the item follows

6. a) only $w(oldfts)$ has changed, so for $F \neq w(oldfts)$ the formula follows from [6a]; if $on_channel(packet(E, F, NR, ACK), peer(P))$ and $F = w(oldfts)$ were both true, then $peer(arrived)(F)$ is true ($\Leftarrow$ [2a, 3])
   b) follows from [6b]
   c) follow from [6c]
   d) $xpa \leq peer(xpf) \leq fts - 1$ ($\Leftarrow$ [5]) and $fts \neq xpa$ ($\Leftarrow$ [1]) so $peer(xpf) \neq fts$, hence the item follows

7. $mode = sending(F, fts - 1)$ is initiated so the item follows

Send event

1. follows from [1]

2. follows from [2]

3. follows from [3]

4. follows from [4]

5. follows from [5]

6. $on_channel(packet(E, F, NR, ACK), peer(P))$ is initiated;
   a) for this packet is $F = w(NR)$ by the event's precondition, for other packets it follows from [6a]
   b) $peer(xpa) - 1 \le ACK \le xpf - 1$ follows from $cnext(ACK, xpf)$
   c) $\forall E' : on_channel(packet(E', F', NR', ACK'), peer(P)) \to E' < E$ so for these we must prove $peer(xpa) - 1 \le ACK' \le ACK \le xpf - 1$ which follows from $cnext(ACK, xpf)$ and [6b] applied to the other packets
   d) for this packet is $NR = fts - 1$ by [7], for other packets the item follows from [6d]

7. follows from $mode \neq sending$

Receive event

We say the arrived packet is $packet(E', F, NR, ACK)$. Several items in the proof for this event use the results of other items: (4) uses (6), (3) uses (2) and (5). This is no problem as there are no cycles.

1. if $net_enabled$ is initiated, then $oldxpa \le ACK < fts$, so $oldxpa <$ $xpa \le fts$, from which the item follows; otherwise, xpa is not changed and the item follows from [1].

2. follows from [2]

3. using item (5) proven below: from $xpa \le peer(xpf) \le fts$ it follows that $xpa - 1 \le peer(xpf) - 1 \le fts - 1$, so applying (2a) it follows that $w(xpa - 1) \le w(peer(xpf) - 1)$; also, from $peer(4)$ proven below it follows that $\forall F' : F' \le w(peer(xpf) - 1) \to peer(arrived)(F')$, therefore $\forall F' : F' \le w(xpa - 1) \to peer(arrived)(F')$

4. it suffices to prove $arrived(peer(w(oldxpf)))$. Because of $NR = oldxpf$ (precondition) and (6a), this is trivially true.

5. from $oldxpa - 1 \leq ACK \leq peer(xpf) - 1$ ($\Leftarrow$ [6b]) it follows that $oldxpa \leq ACK + 1 = xpa \leq peer(xpf)$, which combined with $oldxpa \leq peer(xpf) \leq fts$ ($\Leftarrow$ [5]) yields $oldxpa \leq xpa \leq peer(xpf) \leq fts$, which implies $xpa \leq peer(xpf) \leq fts$

6. a) follows from [6a]
   b) for all *on_channel* packets $packet(E', F', NR', ACK')$ sent by the peer process, it holds that $E < E'$ by the order preservation constraint on packets, and therefore $ACK = xpa - 1 \leq ACK' \leq peer(xpf - 1)$ ($\Leftarrow$ [6c])
   c) follows from [6c]
   d) follows from [6d]

7. follows from $mode \neq sending$

Net_receive event

1. follows from [1]
2. follows from [2]
3. follows from [3]
4. follows from [4]
5. follows from [5]
6. follows from [6]
7. follows from $mode \neq sending$

Timer event

1. follows from [1]
2. follows from [2]
3. follows from [3]
4. follows from [4]
5. follows from [5]
6. follows from [6]
7. follows from [7]

Disturbance event

1. follows from [1]
2. follows from [2]
3. follows from [3]
4. follows from [4]
5. follows from [5]
6. follows from [6]
7. follows from [7]

Failure event

1. follows from [1]
2. follows from [2]
3. follows from [3]
4. follows from [4]
5. follows from [5]
6. follows from [6]
7. follows from [7]

Peer net_send event

1. follows from [1]
2. follows from [2]
3. follows from [3]
4. $\forall F' : F' \leq peer(oldw)(xpf - 1) \leftrightarrow arrived(F')$ ($\Leftarrow$ [4]) so it suffices to prove that $peer(oldw)(xpf - 1) = peer(w)(xpf - 1)$. Now, as only peer(w)(peer(oldfts)) is modified, this amounts to proving that $peer(oldfts) \neq xpf - 1$.
   From $peer(xpa) \leq xpf \leq peer(oldfts)$ ($\Leftarrow peer[5]$) and the fact that $peer(fts)+1 = peer(xpa)$ ($\Leftarrow peer[1]$ and precondition) it follows that $peer(xpa) - 1 \leq xpf - 1 < peer(oldfts)$ which implies $peer(oldfts) \neq xpf - 1$

5. follows from [5]
6. follows from [6]
7. follows from [7]

Peer send event

1. follows from [1]
2. follows from [2]
3. follows from [3]
4. follows from [4]
5. follows from [5]
6. follows from [6]
7. follows from [7]

Peer receive event

1. follows from [1]
2. follows from [2]
3. follows from [3]
4. follows from [4]
5. say the arrived packet is $packet(E', F, NR, ACK)$;
   from $xpa \leq peer(oldxpf) \leq fts$ ($\Leftarrow$ [5]) it follows that

   - if $NR = peer(oldxpf)$ then $xpa \leq NR = peer(xpf) - 1 \leq fts$ but since $NR \neq fts$ ($\Leftarrow$ [6d] $\wedge NR = peer(oldxpf)$) it follows that $xpa \leq peer(xpf) - 1 < fts$, and therefore $xpa < peer(xpf) \leq fts$
   - if $NR \neq peer(oldxpf)$ the item follows from $xpf = oldxpf$
6. follows from [6]
7. follows from [7]

Peer net_receive event

1. follows from [1]
2. follows from [2]
3. follows from [3]
4. follows from [4]
5. follows from [5]
6. follows from [6]
7. follows from [7]

C.2 Proof of the Protocol's Correctness

From the invariant, we can prove that frames arrive in the correct order (more precisely: the arrival of a frame implies that all previous frames have arrived), that they arrive only once, and that each arrived frame has been sent by the other process. We keep using the shorter notation of this appendix; in particular we write $peer(receive)(F,T)$ and $peer(net_send)(F,T)$ to denote a *receive* resp. *net_send* event of P's peer process at T. The first formula to be proven is then:

$$peer(receive)(F,T) \rightarrow \forall F' < F : \exists T' < T : peer(receive)(F',T').$$

Proof:
Because of (6a) and the receive preconditions,

$$F = w(NR) = w(peer(xpf)).$$

Using (5) we find

$$xpa - 1 \leq peer(xpf) - 1 \leq fts - 1.$$

Then we use (2c) : if $fts = peer(xpf)$ then $\neg\exists F : w(peer(xpf) - 1) < F < T$ so no receive event is possible; therefore

$$xpa \leq peer(xpf) \leq fts - 1.$$

From this it follows using (2a) and (2c) that

$$\forall F' : w(peer(xpf) - 1) < F' \leq w(peer(xpf)) \leftrightarrow peer(receive)(F',T)$$

and using $peer(4)$ we know that

$$\forall F' : F' \leq w(peer(xpf) - 1) \leftrightarrow peer(arrived)(F')$$

The last two lines amount to (after T) :

$$\forall F' : F' \leq F(peer(xpf)) \leftrightarrow peer(arrived)(F')$$

which proves the formula. □

The second formula to be proven must ensure that each frame arrives only once. In other words, whenever a process enters receiving mode for a frame, this frame may not have arrived before:

$$initiates(E, attribute(P, mode, receiving(F))) \rightarrow \neg arrived(F)$$

Proof:
Using the completion of the only clause for initiation of a receiving mode, we find that from $initiates(E, attribute(P, mode, receiving(F)))$ it follows that

$$act(E, receive(P, packet(E', F, xpf, ACK)))$$

hence also the preconditions of this action must be satisfied:

$$mode = input$$
$$on_channel(packet(E', F, xpf, ACK), P)$$

Using $peer(6d)$ we then find that $xpf \neq peer(fts)$. On the other hand, from $peer(2a)$ it follows that

$$peer(w)(X - 1) < peer(w)(X) \leftarrow X \neq peer(fts)$$

hence

$$peer(w)(xpf - 1) < peer(w)(xpf)$$

Using (4) we then immediately obtain

$$\neg arrived(peer(w)(xpf))$$

which proves the second formula. □

Finally, we need to prove that each frame received on one side was generated on the other side. Or:

$$initiates(E, attribute(P, mode, receiving(F))) \rightarrow$$
$$\exists T' < T : peer(net_send)(F, T').$$

Proof:
We assume that $initiates(E, attribute(P, mode, receiving(F)))$ and prove that $\exists T' < T : peer(net_send)(F, T')$. Like in the previous proof, we find that

$$act(E, receive(P, packet(E', F, xpf, ACK)))$$

and hence

$$mode = input$$
$$on_channel(packet(E', F, xpf, ACK), P)$$

From $peer(6a)$ it follows that then $F = peer(w)(xpf) \vee arrived(F)$. We also know from the previous proof that in this case $\neg arrived(F)$, hence it follows that $F = peer(w)(xpf)$. This is only possible if F was the initial value of $peer(w)(xpf)$ or if at some earlier time T' the value of $peer(w)(xpf)$ was initiated to F. In the former case, we know that $-n \leq F \leq -1$ by the initial assumptions, which is — also by the initial assumptions — in contradiction with $\neg arrived(F)$. The latter case can only be achieved by an event $peer(net_send)(F, T')$, so $\exists T' < T : peer(net_send)(F, T')$ follows immediately. □

C.3 Deadlock-freeness of the Protocol

We can prove a number of other interesting properties of the protocol, for example deadlock-freeness. As indicated in Chapter 6, deadlock-freeness can be defined as the property that at all times the precondition of some action is true. More specifically we will here prove that at all times, for each process P, at least one of the following actions is possible: *send*, *net_send*, *timer_rings*, *receive* or *net_receive*. In this way we guarantee that at all times each process can create a useful event in the communication.

Proof:
The proof is easiest based on the modes we have defined in the protocol. We will prove that in each mode always at least one of the above events is possible. Since the process is always in exactly one mode, the desired result follows.

- If the process is in mode $sending(F, NR)$, the precondition of *send* shows that a *send* event of a packet $packet(E, F, NR, xpf - 1)$ is possible.
- If the process is in mode $receiving(F)$, the precondition of *net_receive* of frame F is satisfied.
- If the process is in mode $retransmitting(NR)$, a *send* event for a packet $packet(E, w(NR), NR, xpf - 1)$ is satisfied.

- If the process is in *input* mode, always at least one of a *net_send* or a *timer_rings* event is possible: a *timer_rings* event is possible if $xpa \neq fts$, a *net_send* event if *net_enabled*. The latter condition is equivalent to $\neg cnext(fts, xpa)$ by (1), hence at least one of the events can occur unless $xpa = fts$ and $fts = xpa - 1$, which is impossible.

□

Note that whenever a *timer_rings* event occurs, the process enters *retransmitting* mode and only *send* events are possible: hence there is also no risk that all a process will do is have its timers ring.

Despite all of the properties we have proven, there is no guarantee that eventually frames will successfully reach the other side: indeed, an ever-failing channel defeats even the most secure and robust protocols in that respect.

Appendix S

Samenvatting

S.1 Inleiding

De jongste decennia hebben de groei in complexiteit en vereiste flexibiliteit van programmatuur aanleiding gegeven tot het ontstaan van nieuwe programmeerparadigma's. Om de complexiteit onder controle te houden, staan nieuwe programmeertalen dichter bij de menselijke programmeur en steeds verder van de laag-niveau instructies die inwendig in de computer worden gebruikt. Om de vereiste flexibiliteit te bieden, verschuift de programmeertaak van het beschrijven van een oplossing voor een specifiek probleem naar het beschrijven van de relevante gedeelten van het probleemdomein. Zowel het object-georiënteerde paradigma als declaratieve talen in functioneel en logisch programmeren vormen belangrijke stappen in die richting.

Deze trend is zichtbaar bij programmatuur in het algemeen, maar de nood aan flexibiliteit is het meest uitgesproken in het gebied van de kunstmatige intelligentie, waar computers moeten functioneren als steeds autonomere agenten in een veranderende omgeving zonder menselijke tussenkomst. Voor toepassingen in dit domein verandert de taak van de "programmeur" van een klassieke programmeertaak in een opdracht om de aanwezige informatie voor te stellen zodanig dat de agent die op verschillende — intelligente — manieren kan gebruiken. Om dit te bereiken moet aan twee cruciale vereisten voldaan zijn: enerzijds zijn declaratieve talen noodzakelijk die toelaten informatie voor te stellen op een correcte, natuurlijke en bondige manier. Anderzijds moeten voor deze talen flexibele algemene procedures bestaan die de agenten toelaten een wijd gamma van taken uit te voeren. Dit proefschrift concentreert zich op het aspect van correcte kennisrepresentatie. Te gepasten tijde worden algoritmische aspecten kort

besproken, maar deze algoritmen moeten slechts gezien worden als theoretische modellen, niet als efficiënte implementaties.

Klassieke logica is vermoedelijk de best gekende kennisrepresentatietaal. Ze is algemeen toepasbaar en heeft een precieze, natuurlijke semantiek. Deze declaratieve aard van logica maakt het mogelijk op een vrij eenvoudige manier na te gaan of bepaalde delen van een logische voorstelling correct zijn of niet. Verschillende gedeelten kunnen volledig onafhankelijk van elkaar worden gecontroleerd. De ontwikkeling van automatische stellingenbewijzers en meer uitgebreide procedures voor het redeneren over logische theorieën heeft aanleiding gegeven tot logische programmeertalen, waarin probleemdomeinen declaratief kunnen worden voorgesteld en specifieke problemen op een vrij efficiënte wijze opgelost door deductie op de logische theorie.

Als kennisrepresentatietaal is een logische programmeertaal zowel sterker als zwakker dan klassieke eerste-orde logica. Logisch programmeren is sterker door de aanwezigheid van een impliciete *"gesloten wereld"-aanname*: intuïtief uitgedrukt gaat men er in logisch programmeren van uit dat al wat niet vermeld wordt, niet waar is. In de gewone omgangstaal is dit een heel normale veronderstelling: bijvoorbeeld, als men vertelt dat zich op een bepaalde tafel drie blokken A, B en C bevinden en dat A op B staat, dan neemt de luisteraar in het algemeen aan dat er geen andere blokken op de tafel staan en dat C zich niet op of onder A of B bevindt. De gesloten wereld-aanname laat toe enkel aan te geven wat waar is, zonder ook nog eens expliciet toe te voegen wat allemaal niet waar is (wat in het bijzonder bij een oneindig domein ook vaak volstrekt onmogelijk kan zijn). Door deze aanname kunnen in logisch programmeren concepten worden voorgesteld die in eerste-orde logica niet voor te stellen zijn. Anderzijds maakt diezelfde aanname logisch programmeren ongeschikt voor het voorstellen van gedeeltelijke, onvolledige informatie: door de gesloten wereld-aanname is het domein noodzakelijk eenduidig bepaald. In het bovenstaande voorbeeld zou men zo niet kunnen voorstellen dat de positie van C ten opzichte van die van A en B onbekend is: als niet expliciet is gezegd dat C op A of op B staat, wordt aangenomen dat het zeker niet op een van deze blokken staat.

Aangezien men slechts zelden over echt volledige kennis over een bepaald probleemdomein beschikt, is de kracht van logisch programmeren ook zijn grootste nadeel, in het bijzonder omdat onvolledig gekende domeinen veruit de meest interessante zijn. Als een volledige specificatie is gegeven, ligt alles vast, zodat de enige interessante vorm van redeneren over een degelijke theorie deductie is, m.a.w. controleren of een bepaalde uitspraak waar is of niet. Voor een theorie met ongespecificeerde gedeelten zijn ook andere redeneerparadigma's interessant, in het bijzonder abductie en modelgeneratie.

Zo genereert een abductieve redeneervorm hypothesen over het ongespecificeerde deel van de theorie om bepaalde observaties te verklaren. Modelgeneratie is een bijzondere vorm van abductief redeneren, waarin een model voor de volledige verzameling van gegevens wordt gegenereerd.

Open logisch programmeren (OLP) combineert de voordelen van logisch programmeren met die van klassieke eerste-orde logica door deze twee formalismen te integreren. Dit maakt OLP geschikt voor het voorstellen van een heel brede klasse van probleemdomeinen waarin gedeelten volledig gekend en andere gedeelten onvolledig gekend mogen worden verondersteld. Bovendien laten algemene procedures voor open logische programma's toe een groot gamma van zowel deductieve als abductieve taken uit te voeren op een bepaalde theorie, op die manier vermijdend dat dezelfde informatie op een aantal verschillende manieren moet worden gecodeerd voor verschillende taken.

Syntactisch bestaat een open logisch programma uit twee delen. Een eerste deel heeft de vorm van een logisch programma: een verzameling programmaregels van de vorm

$$A \leftarrow B_1, \ldots, B_m.$$

waarbij A een atomaire logische formule (een atoom) is en alle B_i literals, i.e. atomen of negaties van atomen. Het tweede deel is een algemene eerste-orde logische theorie. De betekenis van een logisch programma wordt gegeven door een semantiek die de gesloten wereld-aanname formaliseert voor het gedeelte bestaande uit de programmaregels maar *niet* voor het eerste-orde logica deel (meer precies: alleen voor de predikaten die als gedefinieerd gedeclareerd zijn, niet voor de andere, "open" of ongedefinieerde predikaten). Deze formalisatie kan op verschillende manieren gebeuren, en veel verschillende semantieken bestaan. Voor een eenvoudige klasse van programma's kan de gesloten wereld-aanname worden weergegeven door predikaatvervollediging: als bijvoorbeeld

$$A \leftarrow B_1, \ldots, B_m.$$

de enige definitieregel is voor A, komt zijn betekenis volgens deze semantiek overeen met die van de eerste-orde formule

$$A \leftrightarrow B_1, \ldots, B_m$$

In het algemeen is de formalisatie heel wat complexer, maar we laten de details hier achterwege. Intuïtief is de meest precieze algemene benadering van de gesloten wereld-aanname het lezen van de programmaregels als een inductieve definitie.

Voor het redeneren over OLP-theorieën bestaat een algemene procedure, genaamd SLDNFA, die verschillende redeneervormen (deductieve, abductieve of combinaties van beide) ondersteunt.

S.2 Verband van OLP met Terminologische Talen

In een eerste bijdrage in dit proefschrift tonen we een sterk verband aan tussen OLP en terminologische talen (tegenwoordig bekend onder de naam "Description Logics"). Terminologische talen krijgen veel aandacht in het onderzoek naar kennisrepresentatie en worden gebruikt in nogal wat expertsystemen. Aan de basis van deze talen ligt de observatie in [13] en [11] dat een expertsysteem (en een kennisrepresentatietaal in het algemeen) in staat moet zijn twee essentieel verschillende soorten informatie te beschrijven: enerzijds terminologische, definitionele informatie, bestaande uit definities van een aantal centrale begrippen in termen van meer primitieve begrippen (bijvoorbeeld "Een vader is een ouder die mannelijk is") en anderzijds assertionele informatie over actuele objecten in het probleemdomein (zoals "Jan is de vader van Mia"). Geen van de in 1980 bestaande kennisrepresentatietalen bleek in staat deze twee soorten informatie op de juiste manier te behandelen, wat aanleiding gaf tot de hybride taal KRYPTON en zijn latere opvolgers, de terminologische talen.

In terminologische talen zijn de belangrijkste concepten klassen (*concepts*) en relaties (*roles*). De informatie in een terminologische taal is opgesplitst in twee modules, die elk in een afzonderlijke taal worden weergegeven. Deze modules zijn de T-Box of terminologische component en de A-Box of assertionele component. De A-Box bevat formules van de vorm $a : C$ of aRb, met als betekenis respectievelijk dat a een object is van klasse C en dat b in relatie R staat tot a. De T-Box bevat klassendefinities van de vorm $C == F$, met C een klassensymbool en F een klassenbeschrijving. Afhankelijk van de specifieke terminologische taal zijn verschillende constructies toegelaten als beschrijving van een klasse in de T-Box. De meest eenvoudige constructoren zijn $\sqcup$ (unie), $\sqcap$ (doorsnede), $\neg$ (complement). Verder bestaan bijvoorbeeld constructoren als $\exists$ ($\exists R.C$ is de klasse van objecten x waarvoor minstens één y bestaat zodat xRy en $y : C$), $\forall$ en numerieke beperkingen als $\leq nR$ (de klasse van objecten waartoe hoogstens n objecten in relatie R staan) en $\geq nR$. De jongste jaren worden geleidelijk meer geavanceerde constructoren toegevoegd, onder andere constructoren die de expressiviteit van eerste-orde logica overstijgen.

Een sterk punt van terminologische talen, naast het feit dat ze met

zowel terminologische als assertionele informatie overweg kunnen, is dat efficiënte procedures zijn ontworpen voor elke taal. Complexiteitsanalyse en ontwerp van optimale procedures vormen tegenwoordig het belangrijkste onderzoeksdomein binnen terminologische talen.

We hebben aangetoond dat de bestaande terminologische talen kunnen worden geïnterpreteerd als deeltalen van OLP: voor een aantal terminologische talen hebben we de overeenkomstige OLP-deeltaal precies vastgelegd en de equivalentie bewezen. De overeenkomst is duidelijk op verschillende vlakken: klassen in een terminologische taal komen overeen met unaire predikaten, relaties met binaire predikaten. De A-Box komt overeen met het eerste-orde logica gedeelte van OLP. Zo komt bijvoorbeeld de A-Box formule $a : C$ overeen met de logische formule $C(a)$. De T-Box komt overeen met de logische programmaregels: zo wordt de definitie $C == \exists R.D$ vertaald naar de regel $C(X) \leftarrow R(X,Y), D(Y)$. Voor een aantal andere constructoren is de vertaling iets complexer, maar ze kan steeds worden opgesplitst in voor de hand liggende stappen.

Naast deze declaratieve overeenkomst hebben we ook aangetoond dat de procedures die gebruikt worden voor het redeneren over terminologische talen, gezien kunnen worden als gespecialiseerde instanties van de SLDNFA-procedure die voor OLP gebruikt wordt. We hebben voor een voorbeeldtaal aangegeven welke selectieregel in SLDNFA een equivalente procedure oplevert voor de overeenkomstige deeltaal. Deze zou meteen een optimale procedure opleveren, overeenkomstig de resultaten in terminologische talen.

De overeenkomsten tonen aan dat OLP voldoet aan de belangrijke eis zowel terminologische als assertionele informatie te kunnen voorstellen, en dus een volwaardige kennisrepresentatietaal is. Bovendien blijkt dat de expressiviteit van OLP veel groter is dan die van de bestaande terminologische talen, die slechts overeenkomen met heel beperkte deeltaaltjes van OLP. Ook merken we dat de recente uitbreidingen aan terminologische talen geleidelijk naar grotere deeltalen van OLP evolueren. Op een aantal vlakken is nog onduidelijk welke semantiek voor bepaalde constructies zal worden gedefinieerd. Met het werk in dit proefschrift hopen we een sterk argument te geven voor semantieken die gebaseerd zijn op OLP. Belangrijk voor logisch programmeren is anderzijds de mogelijkheid om tot meer efficiënte procedures te komen, eventueel via een integratie met terminologische talen of door het overnemen van de belangrijkste technieken.

S.3 De Rol van Tijd in Kennisrepresentatie

De voorgaande studie toont de theoretische mogelijkheden van OLP voor kennisrepresentatie aan. In de rest van dit proefschrift complementeren we dit door OLP aan te wenden in zowel praktische als open theoretische problemen op het vlak van kennisrepresentatie.

Het spreekt vanzelf dat kennisrepresentatie in de praktijk een erg probleemdomein-afhankelijke taak is. Desondanks bestaan er ook nog steeds een aantal algemene open vragen, relevant voor grote klassen van toepassingen. Een van de belangrijkste van deze vragen, en zoals blijkt een van de moeilijkste, is hoe men correct rekening kan houden met *tijd*. Dit is van belang in elk probleemdomein dat in zekere zin dynamisch is, dat m.a.w. veranderingen ondergaat of kan ondergaan. Dat hierin zowat alle interessante reële probleemdomeinen zijn vervat, is duidelijk als men bedenkt dat in het bijzonder de agent zelf, die in het domein zijn taken uitvoert, al een belangrijke oorzaak van veranderingen is. Dit maakt dat een correcte voorstelling van dynamische probleemdomeinen van uitzonderlijk belang is.

Het typische aan dynamische domeinen is dat een onderscheid gemaakt moet worden tussen de toestand van het domein op verschillende tijdstippen. Er zijn meestal sterke verbanden tussen toestanden op verschillende dicht bij elkaar gelegen tijdstippen, in het bijzonder door de wet van de inertie: in het algemeen verwachten we dat iets dat geen reden heeft om te veranderen op een bepaald moment, in een zelfde toestand zal blijven. Een agent moet dit "weten", maar het mag niet de taak zijn van een gebruiker die een bepaald domein formaliseert om voor elk tijdstip exhaustief op te sommen wat allemaal onveranderd blijft: een goed representatiesysteem moet toelaten correct af te leiden wat wel en wat niet verandert uit een aantal bondige en intuïtieve regels zoals die typisch door een mens worden gegeven. Met andere woorden, een gespecialiseerde vorm van gesloten wereld-aanname, die de wetten van de tijd in rekening brengt, is vereist. Sinds de jaren '60 wordt een aanzienlijk deel van het onderzoek naar kunstmatige intelligentie besteed aan het zoeken naar een oplossing voor dit centrale probleem, het zogenaamde "frame-probleem".

Het frame-probleem, geïdentificeerd door McCarthy en Hayes in [72], wordt beschouwd als bestaande uit drie deelproblemen. Ten eerste is dat het hoger beschreven inertieprobleem, of het precies bepalen van wat wel en niet verandert ten gevolge van een actie met gegeven bondige effectregels. Een tweede deelprobleem is het kwalificatieprobleem, dat we hier definiëren als het bepalen van de voorwaarden waaronder een actie kan optreden en met welke eventuele gevolgen, eventueel in uitzonderlijke toestanden. Het derde probleem is het ramificatieprobleem, of het bepalen van eventuele onrechtstreekse effecten van een actie.

Naast het frame-probleem zijn er nog een aantal andere belangrijke aandachtspunten in temporeel redeneren, sommige interagerend met het frame-probleem en het nog bemoeilijkend en andere die er loodrecht op staan. Die aandachtspunten zijn bijvoorbeeld het voorstellen van gelijktijdige acties, indeterminisme, onvolledige informatie over optredende acties, effecten die met vertraging optreden en continue verandering. Daarnaast zijn er ook nog een aantal meer fundamentele discussiepunten, zoals wat de topologie van tijd zelf zou moeten zijn (bijvoorbeeld lineair of met vertakkende toekomst, continu of discreet).

Er zijn heel wat formalismen en nog veel meer varianten van deze formalismen voorgesteld om enkele of alle van de bovengenoemde problemen aan te pakken. Het meest courante formalisme is de Situation Calculus, geïntroduceerd in [72]. Een ander vaak gebruikt formalisme is de Event Calculus, oorspronkelijk ontworpen door Kowalski en Sergot in [59]. In dit proefschrift hebben we geopteerd voor een variant van de Event Calculus, die we formaliseren als een OLP-theorie. Voor een korte motivatie en een vergelijking met de Situation Calculus verwijzen we naar de volgende sectie.

Centraal in de Event Calculus is de notie van *events* of gebeurtenissen. Een event is het optreden van een actie van een bepaalde soort op een bepaald moment. Deze events bepalen tijdsintervallen gedurende dewelke *fluents* (tijdsafhankelijke uitspraken over de toestand van de wereld) waar of onwaar zijn. Hieronder geven we kort de belangrijkste axioma's van de Event Calculus weer. Het centrale axioma bestaat uit de programmaregels

$$\begin{array}{lll} holds(P,T) & \leftarrow & happens(E_1,T_1), T_1 < T, initiates(E_1,P), \\ & & \neg clipped(T_1,P,T). \\ clipped(T_1,P,T) & \leftarrow & happens(E_2,T_2), T_1 < T_2, T_2 < T, \\ & & terminates(E_2,P). \end{array}$$

die kort gezegd uitdrukken dat een fluent P waar is op een bepaald tijdstip T als en alleen als er een event E_1 gebeurd is op een tijdstip T_1 vóór T, dat P heeft geïnitieerd, en als tussen T_1 en T geen enkel event P weer heeft getermineerd.

Naast dit zogenaamde "frame-axioma" bevat de Event Calculus axioma's die opleggen dat tijd een totaal geordende verzameling is, dat een fluent nooit op hetzelfde moment geïnitieerd en getermineerd kan worden, meestal dat er een begin-event *start* bestaat dat alles initieert wat waar is in de begintoestand (die weergegeven wordt door het predikaat *initially*) en soms ook dat geen gelijktijdige acties toegelaten zijn. Deze algemene axioma's worden in het algemeen aangevuld met domein-specifieke regels die de effecten van acties beschrijven (in een aantal programmaregels voor *initiates* en *terminates*), en met scenario-informatie (axioma's in eerste-

orde logica of programmaregels die de begintoestand en de optredende acties in een specifiek scenario beschrijven). Een beroemd voorbeeld is het "Yale Shooting Problem", waarmee in het midden van de jaren '80 het falen van zowat alle tot dan toe voorgestelde oplossingen voor het frameprobleem werd aangetoond. In dit eenvoudige probleemdomein zijn drie acties gedefinieerd, met name wachten (*wait*), een geweer laden (*load*) en schieten (*shoot*). De effecten van deze acties worden weergegeven door de regels

$$\begin{array}{lcl} initiates(E, loaded) & \leftarrow & act(E, load). \\ terminates(E, loaded) & \leftarrow & act(E, shoot). \\ terminates(E, alive) & \leftarrow & act(E, shoot), happens(E, T), \\ & & holds(loaded, T). \end{array}$$

Er zijn geen regels waarin *wait* voorkomt: wachten heeft geen enkel effect. Het typische Yale Shooting scenario bestaat uit een opeenvolging van de acties *load*, *wait* en *shoot*, gegeven een begintoestand waarin het mogelijke slachtoffer in leven is. Dit wordt weergegeven door de programmaregels

$$\begin{array}{ll} happens(start, t_0). & initially(alive). \\ happens(e_1, t_1). & act(e_1, load). \\ happens(e_2, t_2). & act(e_2, load). \\ happens(e_3, t_3). & act(e_3, shoot). \end{array}$$

en de eerste-orde logica axioma's

$$t_1 < t_2 \quad t_2 < t_3$$

In dit scenario is alleen het predikaat $<$ ongedefinieerd (open), alle andere predikaten zijn eenduidig gedefinieerd door programmaregels. In eventuele andere scenario's kan onvolledige informatie over optredende events of de begintoestand voorgesteld worden door de predikaten *happens* of *initially* open te laten. De gedeeltelijke informatie erover kan dan worden weergegeven in het eerste-orde logica gedeelte van de theorie.

Verschillende redeneertaken kunnen van belang zijn op Event Calculustheorieën. De meest courante zijn *projectie*, i.e. het bepalen van de volledige evolutie van de wereld gegeven een begintoestand en een sequentie van acties, *postdictie* of *diagnose*, i.e. het bepalen van een begintoestand die een gekende eindtoestand kan verklaren, en *planning*, i.e. het bepalen van een sequentie van acties die van een gegeven begintoestand naar een gewenste eindtoestand leidt. Projectie is essentieel een deductieve taak, terwijl postdictie en planning abductieve taken zijn. Al deze redeneervormen worden ondersteund door de SLDNFA-procedure.

S.4 Een Integratie van Event en Situation Calculus

Zoals eerder aangehaald zijn Event Calculus en Situation Calculus twee van de meest gebruikte formalismen voor het voorstellen van tijdsafhankelijke domeinen. De oorspronkelijke doelstellingen en onderliggende basisideeën van beide formalismen lagen ver uit elkaar, maar latere versies bleken stilaan meer overeenkomsten te vertonen. De jongste jaren werd een analyse van de verschillen en gelijkenissen van beide formalismen dan ook een onderwerp waar nogal wat aandacht aan werd besteed, zoals in [83] en [57]. In deze sectie presenteren we een meer diepgaande analyse die verder bouwt op de hierboven vermelde, en een aantal nog onbeantwoorde vragen aanpakt. Op basis van deze analyse presenteren we een nieuwe calculus die zowel Event als Situation Calculus veralgemeent, en die een aantal problemen aankan die in geen van beide oorspronkelijke calculi behandeld kunnen worden.

Het basisidee in Situation Calculus is dat hypothetische acties aanleiding geven tot toestandsveranderingen of situatie-overgangen. Dit wordt weergegeven in het volgende frame-axioma:

$$\begin{array}{lcl} holds_in(P, s_0) & \leftarrow & initially(P). \\ holds_in(P, result(A,S)) & \leftarrow & init_s(A,S,P). \\ holds_in(P, result(A,S)) & \leftarrow & holds_in(P,S), \neg term_s(A,S,P). \end{array}$$

dat uitdrukt dat in de beginsituatie s_0 alles waar is dat is opgesomd in *initially*, en dat iets waar is in de situatie $result(A, S)$ (die ontstaat na het uitvoeren van actie A in situatie S) als het werd geïnitieerd door A of als het al waar was in S en niet door A getermineerd werd. In het algemeen wordt aan dit basisaxioma een inductie-axioma toegevoegd dat voorstelt dat de enige bestaande situaties diegene zijn die bereikt kunnen worden uit de begintoestand door het uitvoeren van een eindige sequentie van acties.

Het is in de gegeven formalisatie duidelijk dat wat voorgesteld wordt door het Situation Calculus frame-axioma sterk lijkt op wat in de Event Calculus geformaliseerd wordt. Dit wordt onder andere aangetoond in [57]. Toch blijken er bij nader onderzoek opmerkelijke verschillen te bestaan. Zo blijkt dat in Situation Calculus problemen kunnen worden voorgesteld waarin hypothetische (counterfactuele) uitspraken van de vorm "Was A gebeurd, dan zou B waar geworden zijn (maar A is niet gebeurd)." van belang zijn. Deze soort van uitspraken kan niet worden voorgesteld in Event Calculus. De reden hiervoor ligt in het feit dat in Situation Calculus tijd een vertakkende structuur is (elke hypothetische sequentie van acties bestaat in elk model van de Situation Calculus-theorie). In Event Calculus sluiten verschillende sequenties van events elkaar uit, waardoor redeneren

over niet-actuele sequenties onmogelijk is. Anderzijds blijken bepaalde vormen van hypothetisch redeneren van deze soort ook in Situation Calculus onmogelijk, in het bijzonder wanneer acties niet deterministisch zijn. De Event Calculus op zijn beurt heeft voordelen wat betreft het voorstellen van gelijktijdige acties en continue verandering, doordat het met een reële tijdlijn werkt in plaats van met discrete situaties.

Om de voordelen van beide calculi te verenigen en de verbanden zeer precies aan te tonen, ontwerpen we een nieuwe calculus die aan Event Calculus het vertakkende tijdsaspect van Situation Calculus toe te voegen. Dit gebeurt door de axioma's voor lineaire tijd in Event Calculus te vervangen door

$$\neg((T_1 < T_2) \wedge (T_2 < T_1))$$
$$((T_1 < T_2) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_3)$$
$$((T_1 < T_3) \wedge (T_2 < T_3)) \rightarrow (T_1 < T_2) \vee (T_2 < T_1) \vee (T_1 = T_2)$$

die een vertakkende tijdsstructuur opleggen. Daarnaast definiëren we ook het equivalent van een situatie in Event Calculus, op de volgende manier:

$$\begin{array}{ll} s_0 = & \{T \mid (t_0 < T) \wedge \neg int_events(t_0, T)\} \\ result(A, S) = & \{T \mid \exists T' : T' \in S \wedge event(E, T') \wedge act(E, A) \\ & \wedge (T' < T) \wedge \neg int_events(T', T)\} \end{array}$$

of met andere woorden, een situatie is een verzameling van tijdstippen met een gemeenschappelijk laatste voorafgaand event. We definiëren het volgende verband tussen *holds* en *holds_in*

$$holds_in(P, S) \leftrightarrow \forall T : (T \in S \rightarrow holds(P, T))$$

en een overeenkomstig verband tussen *initiates* (*terminates*) en *init_s* (*term_s*):

$$\forall A, S, P : (init_s(A, S, P) \leftrightarrow \exists E, T :$$
$$(T \in S \wedge event(E, T) \wedge act(E, A) \wedge initiates(E, P)))$$
$$\forall A, S, P : (term_s(A, S, P) \leftrightarrow \exists E, T :$$
$$(T \in S \wedge event(E, T) \wedge act(E, A) \wedge terminates(E, P)))$$

Zowel Event Calculus als Situation Calculus zijn speciale gevallen van de op deze manier verkregen nieuwe calculus. De Event Calculus kan verkregen worden door de tijdsstructuur-axioma's te versterken. De Situation Calculus wordt verkregen als we opleggen dat in elke situatie elke actie optreedt in precies één hypothetische vertakking. Onder die voorwaarde kunnen we immers bewijzen dat het frame-axioma van de Event Calculus (dat ook dat van de nieuwe calculus is) equivalent is met dat van Situation Calculus onder de gegeven definities voor situaties.

Deze resultaten tonen het precieze verband aan tussen Event en Situation Calculus, en tussen tijdstippen in het ene formalisme en situaties in het andere. Bovendien tonen ze aan voor welke soort toepassingen beide calculi tekort schieten. Het nieuwe formalisme maakt zowel hypothetisch redeneren mogelijk voor deterministische acties, wat niet kan in Event Calculus, als voor niet-deterministische acties, wat in geen van beide oorspronkelijke calculi kan. Het blijkt anderzijds dat de mogelijkheid tot redeneren over hypothetische acties het enige voordeel vormt van Situation Calculus ten opzichte van Event Calculus, dat anderzijds gemakkelijker kan worden uitgebreid voor toepassingen met continue verandering en gelijktijdige acties. Om deze reden verkiezen we Event Calculus te gebruiken als algemeen formalisme in de rest van dit proefschrift.

S.5 Kennisrepresentatie in OLP Event Calculus

In deze sectie breiden we de OLP Event Calculus uit of passen het formalisme toe in gebieden die niet tot het oorspronkelijk toepassingsdomein binnen de kunstmatige intelligentie behoren. Op die manier illustreren we de flexibiliteit van het formalisme en de bruikbaarheid ervan voor algemene reële toepassingen die de uiterst kleine voorbeeld-applicaties in het meer fundamentele onderzoek overstijgen.

S.5.1 Voorstellen van Continue Verandering

In deze bijdrage formaliseren we een uitbreiding van OLP Event Calculus om continue verandering voor te stellen. In tegenstelling tot de meeste benaderingen tot dit probleem eisen we niet dat deze veranderingen exact gekend zijn of kunnen berekend worden in functie van de tijd. We gaan er immers van uit dat in de praktijk vaak de enige kennis die aanwezig is over een veranderende variabele (b.v. het waterniveau in een vollopende tank), is dat zijn waarde toeneemt, eventueel dat die snel of traag toeneemt. Slechts heel zelden zal men in een reëel probleem een precieze functie kennen die de variabele beschrijft. Toch kan vaak ook uit de heel vage informatie dat bijvoorbeeld een waarde aan het toenemen is, al heel wat worden afgeleid. Gebruik makend van de mogelijkheden van OLP om onvolledige kennis voor te stellen, ontwerpen we een uitbreiding voor de Event Calculus die met dit soort informatie kan werken.

Een belangrijk aandachtspunt hierbij is dat een onderscheid gemaakt moet worden tussen de verandering zelf en eventuele verschillende invloe-

den die samen de verandering veroorzaken. Zo zullen bijvoorbeeld drie geopende kranen een positieve invloed hebben op een veranderend waterniveau, terwijl twee open afvoergaten een negatieve invloed hebben. De reële verandering van het waterpeil ontstaat door combinatie van deze vijf invloeden.

We stellen de volgende axioma's voor, in een stijl die sterk lijkt op die van het Event Calculus frame-axioma, om de invloeden te beschrijven:

$$\begin{array}{lll} influence(I,P,S,T) & \leftarrow & happens(E_1,T_1),\ T_1 < T, \\ & & init_influ(E,I,P,S), \\ & & \neg influ_clipped(I,T_1,P,T). \\ \\ influ_clipped(I,T_1,P,T) & \leftarrow & happens(E_2,T_2),\ T_1 < T_2,\ T_2 < T_1, \\ & & term_influ(E_2,I,P,S). \\ influ_started(I,T_1,P,T) & \leftarrow & happens(E_2,T_2),\ T_1 < T_2,\ T_2 < T_1, \\ & & init_influ(E_2,I,P,S). \\ \\ influ_changed(T_1,P,T_2) & \leftarrow & influ_clipped(I,T_1,P,T_2). \\ influ_changed(T_1,P,T_2) & \leftarrow & influ_started(I,T_1,P,T_2). \end{array}$$

Hierin stelt $influence(I,P,S,T)$ voor dat een bepaalde invloed I op fluent P aanwezig is op tijdstip T, en dat dit een invloed is van soort S. Verschillende soorten invloeden kunnen gedefinieerd worden, elk met hun eigen karakteristieke eigenschappen. Als eenvoudig voorbeeld maken we hier alleen onderscheid tussen positieve en negatieve invloeden, aangeduid door de soorten $+$ en $-$. We stellen dat deze invloeden ook aanleiding kunnen geven tot drie soorten verandering: een positieve (aangeduid door $+$), een negatieve ($-$) en een waarvan de richting onbepaald en mogelijk variabel is ($?$). Dit wordt weergegeven door de volgende definitie voor *cont_change*:

$$\begin{array}{lll} cont_change(P,+,T) & \leftarrow & influence(I,P,+,T),\ \neg any_influ(P,-,T). \\ cont_change(P,-,T) & \leftarrow & influence(I,P,-,T),\ \neg any_influ(P,+,T). \\ cont_change(P,?,T) & \leftarrow & influence(I,P,+,T),\ influence(J,P,-,T). \\ \\ any_influ(P,Sort,T) & \leftarrow & influence(J,P,Sort,T). \end{array}$$

waarbij $cont_change(P,S,T)$ betekent dat fluent P op tijdstip T onderhevig is aan een verandering van soort S. De specifieke waarde van een fluent wordt tijdens een periode van verandering bepaald door het ongedefinieerde predikaat *state_in_change*: $state_in_change(P,X,T)$ zegt dat P de waarde X heeft op tijdstip T. Dit wordt weergegeven door

$$\begin{array}{lll} holds(val(P,X),T) & \leftarrow & cont_change(P,Sort,T), \\ & & state_in_change(P,X,T). \end{array}$$

Het feit dat *state_in_change* een ongedefinieerd predikaat is geeft aan dat we er geen volledige informatie over hebben. Desondanks kunnen we er wel iets over zeggen afhankelijk van het type van de verandering: zo weten we bijvoorbeeld dat tijdens een positieve verandering de waarde stijgt. Dit soort informatie stellen we voor door axioma's in eerste-orde logica, bijvoorbeeld:

$$[same_change(level, +, T_1, T_2),\ holds(val(level, X), T_1), \\ holds(val(level, Y), T_2),\ T_1 < T_2] \rightarrow X \leq Y$$

voor de bovenstaande uitspraak, waarbij *same_change* aangeeft dat twee tijdstippen in eenzelfde periode van continue verandering liggen. Op gelijkaardige manier kan andere informatie, zoals continuïteit, uniciteit van de waarde op een bepaald tijdstip en dergelijke worden weergegeven. De precieze axioma's hangen af van de soorten verandering die onderscheiden kunnen of moeten worden.

Het is eenvoudig aan te tonen dat ons voorstel verenigbaar is met de bestaande toepassingen van Event Calculus, en dat alle redeneervormen die in Event Calculus ondersteund worden, ook mogelijk zijn in onze uitbreiding. We hebben voorbeelden uitgewerkt van projectie-, diagnose- en planningsapplicaties. De voorbeelden tonen onder andere aan dat zelfs uit de zeer beperkte informatie die we veronderstellen al heel wat kan worden afgeleid, zodat we een kwalitatieve benadering van deze soort zeker niet moeten onderschatten, zelfs niet indien we alleen heel primitieve soorten verandering definiëren.

S.5.2 Een Algemene Voorstelling voor Temporele Kennisbanken

Hier illustreren we hoe de OLP Event Calculus gebruikt kan worden als algemeen kader om temporele kennisbanken voor te stellen op een manier die deze kennisbanken nauw doet aansluiten bij toepassingen die er gebruik van maken. Om de precieze doelstellingen te formuleren hebben we ons gebaseerd op het projectvoorstel in [39]. We wensen een kennisbank waarin we zowel uitspraken kunnen doen over de waarheidswaarden van fluents op afzonderlijke tijdstippen als gedurende tijdsintervallen, en waarin we in het algemeen veronderstellen dat de aanwezige informatie onvolledig is. De formules die we toelaten zijn dan ook uitspraken over fluents op tijdstippen of gedurende intervallen, naast informatie over de relatieve ligging van tijdstippen en intervallen ten opzichte van elkaar en uitspraken over ogenblikken waarop fluents van waarheidswaarde veranderen.

In onze formalisatie beschouwen we de kennisbank als bestaande uit twee modules, essentieel een A-Box en een T-Box zoals in terminologische talen.

De T-Box bevat definities voor de formules die we als gegevens aannemen, in termen van Event Calculus primitieven. Deze definities zien eruit als volgt:

$$
\begin{array}{lll}
holds_at(P,T) & \leftarrow & happens(E_1,T_1),\ initiates(E_1,P), \\
 & & T_1 < T,\ \neg clipped(T_1,P,T). \\
holds_in(P,int(T_1,T_2)) & \leftarrow & interval(T_1,T_2),\ holds_from(P,T_1), \\
 & & \neg clipped(T_1,P,T_2). \\
notholds_in(P,int(T_1,T_2)) & \leftarrow & interval(T_1,T_2), \\
 & & notholds_from(P,T_1), \\
 & & \neg started(T_1,P,T_2).
\end{array}
$$

$$
\begin{array}{lll}
started(T',P,T) & \leftarrow & happens(E'',T''), initiates(E'',P), \\
 & & (T' < T''),(T'' < T). \\
clipped(T',P,T) & \leftarrow & happens(E'',T''), terminates(E'',P), \\
 & & (T' < T''),(T'' < T). \\
holds_from(P,T) & \leftarrow & happens(E,T),\ initiates(E,P). \\
holds_from(P,T) & \leftarrow & happens(E,T),\ holds_at(P,T), \\
 & & \neg terminates(E,P). \\
notholds_from(P,T) & \leftarrow & happens(E,T),\ terminates(E,P). \\
notholds_from(P,T) & \leftarrow & happens(E,T),\ \neg holds_at(P,T), \\
 & & \neg initiates(E,P).
\end{array}
$$

$$
\begin{array}{lll}
on(P,T) & \leftarrow & happens(E,T),\ initiates(E,P), \\
 & & \neg holds_at(P,T). \\
off(P,T) & \leftarrow & happens(E,T),\ holds_at(P,T), \\
 & & terminates(E,P).
\end{array}
$$

aangevuld met definities die de onderlinge ligging van intervallen bepalen, overeenkomstig de relaties voorgesteld in [3], bijvoorbeeld

$$
\begin{array}{lll}
overlaps(int(T_1,T_2),int(T_3,T_4)) & \leftarrow & interval(T_1,T_2), interval(T_3,T_4), \\
 & & T_1 < T_3,\ T_3 < T_2,\ T_2 < T_4. \\
during(int(T_1,T_2),int(T_3,T_4)) & \leftarrow & interval(T_1,T_2), interval(T_3,T_4), \\
 & & T_3 < T_1,\ T_2 < T_4.
\end{array}
$$

$$
\begin{array}{lll}
interval(T_1,T_2) & \leftarrow & happens(E_1,T_1), T_1 < T_2, \\
 & & happens(E_2,T_2).
\end{array}
$$

De A-Box bevat de eigenlijke gegevens in een actuele toestand van de kennisbank, als een verzameling eerste-orde logische formules opgebouwd uit de hierboven gedefinieerde basisformules met behulp van de klassieke ope-

ratoren, bijvoorbeeld

$$
\begin{aligned}
¬holds_in(has(john, book_1), int(t_1, t_2))) \\
&holds_at(p(a), T) \rightarrow holds_at(q(b), T) \\
&\forall T : (holds_at(p, T)) \\
&holds_at(has(X, O), T) \wedge holds_at(has(Y, O), T) \rightarrow X = Y
\end{aligned}
$$

Gegeven de hierboven gedefinieerde kennisbank, kan de SLDNFA-procedure in principe gebruikt worden om de noodzakelijke functionaliteit van zo'n kennisbank te implementeren. Zo is het eenvoudig om de consistentie van de gegevens te controleren en om vragen te beantwoorden, zowel van het type "Is Q consistent met de gegevens ?" als "Volgt Q uit de gegevens ?". Dit onderscheid moet gemaakt worden gezien de kennisbank onvolledige informatie voorstelt, zodat deze twee vragen niet equivalent zijn. Complexe vragen kunnen beantwoord worden door een voorafgaande transformatiestap te gebruiken waarvan de correctheid bewezen is in [68]. Naast deze basisfunctionaliteit ondersteunt SLDNFA ook rechtstreeks het gebruik van de kennisbank voor bijvoorbeeld planning. Dit kan mits een kleine wijziging (het invoeren van acties, die normaal niet voorzien zijn in de kennisbank) op dezelfde manier gebeuren als gebruikelijk in Event Calculus. Daarenboven biedt SLDNFA ondersteuning voor het herstellen van de consistentie van een inconsistente kennisbank.

Belangrijk is in deze bijdrage dat eenzelfde formalisme gebruikt wordt voor een kennisbank en de eventuele toepassingen die er gebruik van maken. Op deze manier voorkomen we nodeloze spraakverwarring bij de interactie van verschillende componenten in een systeem. De voorgestelde algoritmen moeten wel slechts beschouwd worden als theoretische modellen. Ze zijn op zich niet bruikbaar als efficiënte implementaties.

S.5.3 OLP Event Calculus als Protocolspecificatietaal

In de hier beschreven bijdrage gebruiken we de OLP Event Calculus voor protocolspecificatie. Hiervoor worden in het algemeen heel gespecialiseerde talen, zogenaamde procesalgebra's, gebruikt. Deze modelleren een proces als een algebraïsche structuur, opgebouwd uit primitieve deelprocessen met behulp van een aantal sequentie-, keuze- en synchronisatie-operatoren: een proces wordt gemodelleerd als de verzameling van mogelijke opeenvolgingen (*sporen*) van gebeurtenissen die door het proces kunnen worden uitgevoerd of waaraan het kan deelnemen.

Anderzijds is een natuurlijke visie op een proces die van een dynamische structuur, met een inwendige toestand die verandert met de tijd, en met de mogelijkheid tot het uitvoeren van acties en het opmerken van

acties die door andere processen worden uitgevoerd. Vanuit dit perspectief mag verwacht worden dat de OLP Event Calculus heel geschikt is voor het specificeren van procesprotocollen. Dit hebben we concreet onderzocht met als voorbeeld het "sliding window protocol with go-back-n," een communicatieprotocol gesitueerd in de datalink-laag van de OSI-netwerkarchitectuur. Dit protocol heeft als doel een betrouwbare communicatie tussen twee processen te verzekeren gegeven een onbetrouwbare fysische verbinding. Voor een informele specificatie verwijzen we naar [105].

In Event Calculus stellen we een proces voor als een entiteit waaraan een aantal fluent-attributen zijn gekoppeld die zijn interne toestand weergeven. Voor een proces in een communicatieprotocol zijn deze attributen bijvoorbeeld de *mode*, die aangeeft of een proces wacht op invoer of bezig is met het zenden of ontvangen van berichten, en een reeks parameters die helpen bij de boekhouding, bijvoorbeeld de identificatienummers van verzonden maar onbevestigde berichten, het nummer van het volgende verwachte bericht, en parameters die aangeven of nieuwe te verzenden informatie aangenomen kan worden. Vervolgens bepalen we welke soorten events door de processen in een bepaald protocol kunnen worden uitgevoerd of spontaan kunnen optreden (zoals een storing op het kanaal). Voor elk van deze event-types schrijven we de precieze precondities en de effecten van zo'n event op de toestand van het proces en de buitenwereld uit, in de gebruikelijke Event Calculus-stijl als een definitie voor de predikaten *initiates* en *terminates*. Als voorbeeld geven we hier de preconditie en een van de effecten van de aankomst van een nieuw pakket gegevens over het kanaal. De preconditie is dat het pakket dat ontvangen wordt, zich op het kanaal bevindt, en dat het ontvangende proces in invoermode staat.

$$\begin{array}{l} precondition(receive(PROCESS, PACKET), T) \\ \leftarrow \left| \begin{array}{l} holds(attribute(PROCESS, mode, input), T), \\ holds(on_channel(PACKET, PROCESS), T). \end{array} \right. \end{array}$$

Effecten zijn bijvoorbeeld dat het proces overgaat in ontvangstmode en dat het nummer van het verwachte pakket met één wordt verhoogd, op voorwaarde tenminste dat het pakket datgene is dat verwacht werd en dat het niet door storingen is gewijzigd (indien een onverwacht of gewijzigd pakket aankomt, wordt het gewoon genegeerd). Dat wordt uitgedrukt door

de regel

$$
\begin{array}{l}
initiates(E, attribute(PROCESS, xpf, NEXTXPF)) \\
initiates(E, attribute(PROCESS, mode, receiving(FRAME))) \\
\quad \leftarrow \left| \begin{array}{l}
happens(E,T),\ act(E, receive(PROCESS, PACKET)), \\
\neg holds(corrupt(PACKET), T) \\
PACKET = packet(E', FRAME, NR, ACK), \\
holds(attribute(PROCESS, xpf, XPF), T), NR = XPF, \\
cnext(XPF, NEXTXPF).
\end{array} \right.
\end{array}
$$

Op deze manier wordt de volledige specificatie van het protocol opgebouwd.

Deze wijze van specificeren vertoont verrassend grote verschillen met de gebruikelijke stijl in procesalgebra's: wat wordt gespecificeerd zijn algemene eigenschappen van een proces, toestandsvariabelen, en de evolutie daarvan. Hieruit kunnen dan, bijvoorbeeld met SLDNFA, de mogelijke opeenvolgingen van events worden berekend. In procesalgebra's zijn het rechtstreeks die event-sequenties die worden gemodelleerd, en niet de onderliggende eigenschappen. Een gevolg is dat een Event Calculus-specificatie meteen ook bruikbaar is voor andere toepassingen in het gegeven probleemdomein, zoals bijvoorbeeld netwerkbeheer, omdat ze alle gegevens die daarvoor van belang zijn modelleert. In procesalgebra's zijn al deze gegevens wegvereenvoudigd en blijft alleen de strikt noodzakelijke informatie voor protocolspecificatie over. Voor andere toepassingen moet het domein hierdoor opnieuw en op een volstrekt andere manier worden gespecificeerd. Ondanks deze grotere algemeenheid blijkt een Event Calculus-specificatie (in elk geval al voor het bovenstaande protocol) slechts van dezelfde lengte als een overeenkomstige specificatie in een procesalgebra. Aan de andere kant verloopt het automatisch redeneren over een Event Calculus-specificatie, bijvoorbeeld voor het verifiëren van protocoleigenschappen zoals de onmogelijkheid van deadlock-situaties, heel wat minder efficiënt, al is dit ook ten dele te wijten aan de voorlopig nog verre van optimale implementatie van SLDNFA.

S.6 Een Hoog-niveau Representatietaal voor Dynamische Probleemdomeinen

De vorige secties beschreven uitbreidingen van OLP Event Calculus voor gebruik in minder traditionele toepassingsdomeinen van het formalisme. In deze sectie keren we terug naar het klassieke toepassingsgebied van de fundamentele kunstmatige intelligentie: we tonen aan hoe OLP Event Calculus een grote stap voorwaarts kan opleveren in het aanpakken van het frameprobleem, in het bijzonder het ramificatieprobleem.

In dit kader heeft de expressiviteit van OLP minstens evenveel nadelen als voordelen: omdat informatie heel genuanceerd voorgesteld kan worden, is een goede kennis van de precieze nuances van verschillende voorstellingswijzen nodig om te garanderen dat de bedoelde informatie weergegeven wordt. Het is niet realistisch om zo'n gedetailleerde kennis te verwachten van elke gebruiker. Er is een methodologie vereist die precies aangeeft hoe elke bepaalde soort informatie voorgesteld moet worden opdat deze correct wordt geïnterpreteerd door het systeem.

Om die reden ontwerpen we een hoog-niveau taal om dynamische probleemdomeinen voor te stellen, waarin precies wordt vastgelegd hoe welke informatie kan worden weergegeven. Tegelijkertijd dragen we er zorg voor dat alle taalconstructies die verantwoordelijk zijn voor de "goede" expressiviteit van OLP Event Calculus, ook in de nieuwe taal aanwezig zijn. Dit vereist een diepgaande analyse van de soorten probleemdomeinen die we willen voorstellen, gevolgd door een keuze van gewenste taalconstructies.

Doelstellingen voor de taal die we ontwerpen zijn dat we correct de gevolgen van optredende acties kunnen weergeven, zowel directe als in het bijzonder indirecte gevolgen, zowel onmiddellijke als uitgestelde effecten, zowel van opeenvolgende als gelijktijdige acties, zowel van deterministische acties als van acties met indeterministische effecten. Daarnaast willen we ook correct kunnen weergeven onder welke voorwaarden welke acties kunnen optreden. Bovendien willen we dit op een flexibele manier realiseren voor domeinen waarin de begintoestand en de optredende acties en hun volgorde al dan niet volledig gegeven zijn. We noemen onze taal $\mathcal{ER}$, wat ruwweg staat voor "Event-gebaseerde taal voor Ramificaties", vermits indirecte effecten, m.a.w. het ramificatieprobleem, veruit onze belangrijkste zorg zijn.

Een analyse van bestaande voorstellen in de literatuur toont aan dat ramificaties in het algemeen worden beschouwd als sterk verwant aan toestandsbeperkingen (*state constraints*). Dit zijn altijd-geldende verbanden tussen fluents in het domein, bijvoorbeeld voor twee verbonden tandwielen dat ze ofwel allebei draaien ofwel allebei in rust zijn: $turning_1 \leftrightarrow turning_2$. Het verband met ramificaties is in het voorbeeld het volgende: als op èén van de tandwielen een kracht wordt uitgeoefend zodat het begint te draaien, zal door de verbinding een krachtpropagatie plaatsvinden zodat ook het andere tandwiel aan het draaien gaat. Omgekeerd zal ook het stilleggen van een tandwiel het stoppen van het andere voor gevolg hebben.

In de literatuur is aangetoond dat toestandsbeperkingen op zich niet voldoende zijn om precies de bedoelde ramificaties te voorspellen: zo geven beperkingen ook vaak aanleiding tot impliciete precondities voor acties. Bijvoorbeeld, men kan opleggen dat iemand die geen diploma heeft in een

bepaald bedrijf geen manager kan worden: $manager(X) \rightarrow diploma(X)$. Hier is het uiteraard niet de bedoeling dat iemand tot manager wordt gepromoveerd en als neveneffect meteen een diploma krijgt.

Om preciezer aan te geven waar toestandsbeperkingen aanleiding geven tot precondities en waar tot indirecte effecten, zijn zogenaamde causale wetten ingevoerd: een soort toestandsbeperking met impliciete indicatie van de richting waarin effecten kunnen propageren.

Een van de belangrijke bijdragen in dit proefschrift is dat we aantonen dat dit onvoldoende is, in de zin dat indirecte effecten niet altijd afhangen van toestandsbeperkingen, maar eigenlijk manifestaties zijn van fysische of eventueel logische "krachtpropagaties". Toestandsbeperkingen kunnen uit specifieke patronen van dergelijke propagaties ontstaan, zoals in het voorbeeld met de tandwielen hierboven de fysische krachtpropagaties ervoor zorgen dat $turning_1 \leftrightarrow turning_2$ altijd waar blijft. Maar in veel gevallen onstaat er ook helemaal geen specifieke toestandsbeperking uit de indirecte effecten. Het eenvoudigste voorbeeld is dat van een teller in een elektronisch netwerk, dat een aantal gebeurtenissen (voorgesteld door het overgaan van de spanning op een bepaalde plaats in het netwerk van laag naar hoog) telt. Telkens de spanning in die zin verandert, verhoogt de teller met één. Het is duidelijk dat hier geen altijd-geldende relatie tussen gelijktijdige fluents door ontstaat.

Om deze reden voorzien we constructies die indirecte effecten weergeven als onafhankelijke manifestaties van effectpropagaties. We laten toe dat deze regels complex zijn, m.a.w. dat effecten kunnen voortkomen uit willekeurige combinaties van andere effecten, wat aanleiding geeft tot heel bondige beschrijvingen en een eenvoudige behandeling van gelijktijdige acties mogelijk maakt. Daarnaast maken we nog steeds gebruik van toestandsbeperkingen, omdat deze een gemakkelijke hoog-niveau-beschrijving van een domein toelaten, en van de noodzakelijke basisconstructies, zijnde expliciete precondities voor acties en regels voor directe effecten. Tenslotte hebben we nog een aantal eenvoudige constructies nodig voor het voorstellen van scenario-informatie.

Syntactisch zien de voorziene constructies eruit als volgt:

- **direct-effectregels** van de vorm

  $$a \textbf{ causes } l \textbf{ if } F'$$

  met a een actie, l een fluent of negatie van een fluent, en F' een complexe fluentformule. Deze regels geven weer dat l waar wordt telkens a wordt uitgevoerd terwijl F' waar is;

- **afgeleid-effectregels** van de vorm

$$\textbf{initiating } F \textbf{ causes } l \textbf{ if } F'$$

  met l een fluent of negatie van een fluent, en F en F' complexe fluentformules. Deze regels geven weer dat l waar wordt telkens F van onwaar naar waar verandert op een moment waarop F' waar is;

- elke formule op de klassieke manier geconstrueerd uit de predikaten **Holds**, **Happens**, $\leq$ en **Initially** met behulp van $\neg, \wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow$ en $\forall, \exists$.

In de laatste klasse onderscheiden we in het bijzonder

- **toestandsbeperkingen** geschreven in de vorm

$$\forall T : \mathbf{Holds}(F, T)$$

- **precondities** in de vorm

$$\forall T : \mathbf{Happens}(a, T) \rightarrow \mathbf{Holds}(F, T)$$

terwijl andere formules volledige of onvolledige informatie kunnen voorstellen over de waarheidswaarde van fluents op bepaalde tijdstippen, het optreden en de volgorde van acties, en de begintoestand.

De semantiek van een $\mathcal{ER}$-theorie wordt gedefinieerd door de effectregels te lezen als een inductieve definitie voor predikaten **Init** en **Causes**, en de andere formules als in klassieke eerste-orde logica. We laten de wiskundige details hier achterwege. Onze formalisatie laat toe op een correcte manier recursie en lussen in effectregels te behandelen: de inductieve-definitie-semantiek is constructief, zoals effectpropagaties in de realiteit verwacht worden te zijn. Zo zal een effect nooit "veroorzaakt" worden door zichzelf, en worden ook effecten die afhangen van de afwezigheid van andere effecten correct afgeleid. Aan de andere kant laat het eerste-orde logisch gedeelte toe op een flexibele manier volledige of onvolledige informatie voor te stellen over scenario's. Door de wisselwerking van eerste-orde logica voor scenario-informatie en inductieve definities voor effectregels krijgen we een expressieve taal die alle mogelijkheden van OLP Event Calculus biedt.

De hierboven beschreven basistaal is uitgebreid met constructies om ook acties met indeterministische effecten en uitgestelde ramificaties te kunnen voorstellen. Voor het voorstellen van indeterminisme worden regels van de volgende vormen gebruikt:

- **direct-effectregels** van de vorm

$$a \textbf{ causes } D \textbf{ if } F'$$

waarbij D nu een disjunctie van conjuncties van fluent literals is;

- **afgeleid-effectregels** van de vorm

$$\textbf{initiating } F \textbf{ causes } D \textbf{ if } F'$$

met D ook hier een disjunctie van conjuncties van fluent literals

De semantiek van deze regels wordt gedefinieerd door te eisen dat voor elke indeterministische effectregel, minstens één van de regels verkregen door de disjunctie D te vervangen door een van zijn disjuncten, geldig moet zijn op elk ogenblik. We hebben aangetoond dat dit de meest precieze manier is om indeterminisme voor te stellen.

Voor het voorstellen van uitgestelde effecten hebben we geopteerd voor regels van de vorm

$$\textbf{initiating } F \textbf{ if } F' \textbf{ ecauses } e \textbf{ if } F'' \textbf{ persists after } d$$

wat betekent dat het initiëren van F op een moment waarop F' waar is, aanleiding geeft tot het event e, een tijd d na deze gebeurtenis, op voorwaarde dat F'' waar blijft tot op het moment waarop e zou gebeuren. Deze regels definiëren dus dat een initiatie van een complexe formule aanleiding geeft tot een later event, maar laten onrechtstreeks ook toe te definiëren dat een event aanleiding geeft tot een later event of een latere initiatie, of een initiatie tot een latere initiatie: dit kan door de regels te combineren met de gewone effectregels. De semantiek van uitgesteld-effect-regels wordt gedefinieerd door ze te lezen als een inductieve definitie voor **Happens**.

Om de cirkel rond te maken, hebben we een eenduidige vertaling van $\mathcal{ER}$-theorieën naar OLP Event Calculus gedefinieerd, en bewezen dat de resulterende theorie equivalent is met de oorspronkelijke. De vertaling beeldt effectregels af op programmaregels die een definitie vormen voor *initiates* en *terminates*, en andere formules op eerste-orde logische formules. Nietdeterministische regels worden vertaald naar programmaregels met vrijheidsgraad-predikaten, wat onze gebruikelijke aanpak voor indeterminisme is in OLP Event Calculus. Regels voor uitgestelde effecten worden vertaald naar een definitie voor *happens* in termen van vroegere effecten.

In de hierboven gedefinieerde taal $\mathcal{ER}$ hebben we ook onderzocht hoe het onlangs door Thielscher ([109]) geopperde idee kan uitgewerkt worden om zogenaamde *invloedsinformatie* te gebruiken om automatisch effectregels af te leiden uit toestandsbeperkingen. Invloedsinformatie geeft

aan welke fluents een invloed op andere fluents kunnen uitoefenen en dus onrechtstreeks in welke richting effecten kunnen propageren. We hebben Thielscher's voorstel bestudeerd en verbeteringen aangebracht, en grondig de verschillen tussen beide voorstellen besproken. Al bij al blijkt dat invloedsinformatie vaak onvoldoende nauwkeurig is om tot duidelijke juiste resultaten te komen. In onze methode worden dergelijke probleemgevallen automatisch ontdekt en aangegeven. Los daarvan hebben we bewezen dat onze methode voldoet aan een streng correctheidscriterium, dat oplegt dat toestandsbeperkingen alleen hersteld mogen worden met minimale effecten, die allemaal verantwoord moeten zijn door de invloedsinformatie. Wel spreekt het vanzelf dat elke methode die gebruik maakt van invloedsinformatie beperkt is door het feit dat niet alle effectregels voortkomen uit toestandsbeperkingen.

De belangrijkste bijdrage van deze sectie is dat een nieuwe taal is ontworpen waarin de nodige constructies worden aangeboden om het ramificatieprobleem op te lossen in een voor het overige heel algemene context. We behandelen een heleboel hete hangijzers, zoals gelijktijdige acties, indeterminisme, onvolledige informatie en complexe causale wetten, in één coherent raamwerk. Hierin zijn we geslaagd door taalconstructies en semantiek zo te kiezen dat ze zo nauw mogelijk aansluiten bij de menselijke intuïtie, eerder dan bij gebruikelijke aanpakken. Ten gevolge hiervan kunnen we ook een aantal tot nu toe onbehandelde problemen oplossen in hetzelfde kader, zoals lussen in effectregels en ramificaties die los staan van toestandsbeperkingen. We hebben de overeenkomsten en verschillen van onze aanpak aangetoond met uiteenlopende recente voorstellen, waaronder dat van Thielscher ([109]) dat het meest op het onze lijkt en dat we in detail hebben bestudeerd. Voor een aantal andere aanpakken kunnen we dezelfde verschilpunten aanhalen als Thielscher.

S.7 Besluit

De doelstellingen in dit proefschrift waren het bestuderen en toepassen van de mogelijkheden van open logisch programmeren voor kennisrepresentatie, in het bijzonder in een brede klasse van probleemdomeinen die veranderlijk zijn in de tijd, en dit zowel op het vlak van de fundamentele kunstmatige intelligentie als in meer directe toepassingen.

Op taal-theoretisch vlak hebben we aangetoond hoe OLP een veralgemening is van de bestaande terminologische talen, dezelfde basisprincipes hanterend voor kennisrepresentatie maar in een veel expressievere taal.

In de rest van dit proefschrift gebruiken we het formalisme in tijdsafhankelijke probleemdomeinen. Op het vlak van de fundamentele kunstma-

tige intelligentie analyseren we OLP-formalisaties van Situation Calculus en Event Calculus en construeren een nieuw formalisme dat beide veralgemeent. Verder ontwerpen we een expressieve hoog-niveau taal voor dynamische probleemdomeinen, die constructies aanbiedt om het frame- en in het bijzonder het ramificatieprobleem op te lossen, ook in aanwezigheid van mogelijk gelijktijdige en niet-deterministische acties met eventueel uitgestelde effecten, van algemene effectpropagaties, en van mogelijk onvolledige scenario-informatie.

Verder tonen we aan hoe OLP Event Calculus uitgebreid kan worden voor het voorstellen van kwalitatief gekende continue verandering, hoe het een algemeen kader biedt voor het voorstellen van temporele kennisbanken, zowel wat betreft voorstelling als functionaliteit, en hoe het een nieuwe manier aanbiedt om aan protocolspecificatie te doen.

Een belangrijke opdracht voor de toekomst, die nog niet aan de orde was in dit proefschrift, is het werken aan efficiëntere implementaties van de procedures voor OLP. Op dit moment is het gebrek aan efficiëntie immers nog steeds een nadeel ten opzichte van meer gespecialiseerde technieken, bijvoorbeeld in temporele kennisbanken of protocolverificatie, een nadeel dat opweegt tegen de geboden voordelen. Nieuwe projecten die deze efficiëntieproblemen zullen aanpakken, gaan rond de tijd van dit schrijven van start.