Action Logic Programs — How to Specify Strategic Behavior in Dynamic Domains Using Logical Rules

D i s s e r t a t i o n

zur Erlangung des akademischen Grades
Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Diplom-Informatiker Conrad Drescher
geboren am 31.01.1977 in Gräfelfing

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. habil. Michael Thielscher School of Computer Science and Engineering The University of New South Wales Australia
Externer Gutachter:	Prof. Dr. Tommie Meyer Meraka Institute African Advanced Institute for ICT South Africa
Tag der Verteidigung:	19.07.2010

Oxford im März 2011

Contents

Acknowledgements

First of all i want to express my gratitude towards my supervisor Michael Thielscher for his continuous support throughout the years, for all the helpful discussions, and for running such a very nice group. Very special thanks are also due to Franz Baader, Matthias Fichtner, Ozan Kahramanogullari, Hongkai Liu, Carsten Lutz, Maja Milicic, Uwe Petersohn, Stephan Schiffel, Peter Steinke, Hannes Strass, Sebastian Voigt, and Dmitry Tishkovsky — in one way or another they have all contributed substantially to the genesis of this thesis. A big special thanks goes to Tommie Meyer for accepting to be my external reviewer.

1 Introduction

Imagine a software agent situated in some kind of dynamic environment; i.e. there are mutable properties, both under, and beyond the agent's control. Now imagine the agent is trying to achieve some goal in that dynamic domain: The agent is facing an initial state of the world, and via some actions it is trying to transform this state into a world state that satisfies the goal. The agent may directly execute the actions in an online fashion, hoping that it will achieve the goal. But the agent can also search for a sequence of actions that will guarantee that the goal is achieved, i.e. the agent can do offline planning. In both cases the agent has to perform some form of reasoning, both about the state it is currently in, and about the strategy that might lead to the goal. In this thesis we are looking at the problem of how to specify appropriate strategies for this type of agent in both the online and the offline setting.

This "agent" metaphor is quite general: Our agent could be a physical agent, e.g., an autonomous robot, or it could be a software agent, e.g., a broker service on the web trying to compose other web services so that a certain user demand is satisfied. There are two main ingredients for this metaphor to work: First of all, we need a model of the dynamic domain. For this we need to be able to specify what the domain looks like (initially, currently, finally), and we need to know how it evolves once some actions are executed. But the second ingredient is equally important: The model of the dynamic domain is likely to admit a plethora of possible agent behaviors; and it is unlikely that every such behavior can be considered useful (Imagine a robot that just keeps moving back and forth). Thus the second important ingredient for our "agent in a dynamic domain" metaphor is some means to single out the useful behaviors from the possible. It is this task that this work is foremost concerned with.

Now we can imagine many different models of the dynamic domain, and likewise many different strategy selection mechanisms. The only restriction is that a model of the dynamic domain has to be formal and symbolic, in the sense that it can actually be implemented on a computer. Apart from that, the model can be probabilistic, logic-based, ad-hoc, etc.. The strategy can be purely reactive, based on formalizations of beliefs, desires, and intentions, utility-maximizing, etc..

What we will do in this thesis is to use first order logical rules (more specifically, Horn clauses) to specify agent strategies atop of dynamic domains that are likewise modeled in classical first order (and sometimes second order) logic.

One advantage of formal logic is that it is a well-studied formal model. Consequently, we can reason fully formally over properties that apply to the approach as

a whole, to a specific instance, etc.. The formal semantics of logic ensures that we always know precisely what our modeling means — there is no ambiguity. In the case of a more ad-hoc approach the semantics is given by the actual implementation — the program means what it does. As a result a change in the implementation may result in a drastic change in the meaning of the program. The main drawback of a principled logic-based model is that it seems to be hard to master; people naturally seem to prefer the more ad-hoc approaches. This thesis is based on a logic-based modeling, with the hope that it is simple, and intuitive enough for people to find it useful. Extending our work to a mixed logic- and probability-based model is left as a research challenge for future work.

The rest of this introduction is organized as follows: We start with an introduction to logic-based models of dynamic domains. After that we give a brief introduction to the existing approaches to specifying strategic behaviour in dynamic domains (that have been formalized in classical logic). Next we discuss a range of different logics that have been put forth; here we focus especially on their merits with regard to reasoning about dynamic domains. It is here where we will try to motivate our choices wrt. the logic underlying the material presented in this thesis. Finally, we include a guide to the structure of the thesis.

1.1 Reasoning about Actions and Change with Logical Action Calculi

In this section we recall the existing approaches to modeling dynamic domains in classical logic. In particular, this will be the Event, the Fluent, and the Situation Calculus. We will also provide a brief overview of research in planning formalisms: These typically are not directly based on logic — but more often than not they can equivalently be expressed as a fragment of one of the aforementioned action calculi.

1.1.1 Event, Fluent, and Situation Calculus

Let us now introduce the three major classical action calculi. There are three that stick more or less close to classical first order logic: The Event, the Fluent, and the Situation Calculus. We start by recalling the respective basic ideas.

Situation Calculus

The first logic-based formalism for modeling dynamic domains to be introduced has been the Situation Calculus of McCarthy [McCarthy, 1963]. The basic idea introduced there is to extend ordinary logical atoms like, e.g., $\mathrm{On}(\mathrm{Block}_1, \mathrm{Block}_2)$ by an additional argument denoting the time-point when this logical atom holds. The time structure invented for this purpose is that of situations (giving Situation

Calculus its name): S_0 is a term that is used to denote the initial situation. Next we have action terms, like, e.g., $\mathrm{Move}(\mathrm{Block}_1, \mathrm{Table})$, and these are used to build new situations from old ones. To see how this works consider, e.g., the following statement:

$$\mathrm{On}(\mathrm{Block}_1, \mathrm{Table}, \mathrm{Do}(\mathrm{Move}(\mathrm{Block}_1, \mathrm{Table}), S_0)),$$

expressing that a certain block will be located on the table if it has just been moved there.[1] Such an atom expressing time-dependent information is called a fluent. Of course, in the initial, or any other, situation there may be several actions that are applicable. Depending upon the choice of a particular action, a different next situation is obtained — thus situations provide a model of branching time.

Now assume that we axiomatize the effects of Block_1 being moved to the table as

$$\begin{aligned}&\mathrm{Poss}(\mathrm{Move}(\mathrm{Block}_1, \mathrm{Table}), s) \supset \\ &\quad \mathrm{On}(\mathrm{Block}_1, \mathrm{Table}, \mathrm{Do}(\mathrm{Move}(\mathrm{Block}_1, \mathrm{Table}), s)),\end{aligned}$$

meaning that, if we move a block to the table in some situation s, then in the next situation it will be on the table. Furthermore, if we axiomatize the action precondition as

$$\mathrm{Poss}(\mathrm{Move}(\mathrm{Block}_1, \mathrm{Table}), s) \equiv \top,$$

then from this together with the effect axiom we can conclude that

$$\mathrm{On}(\mathrm{Block}_1, \mathrm{Table}, \mathrm{Do}(\mathrm{Go}(\mathrm{Block}_1, \mathrm{Table}), S_0)).$$

But assume we had some more information about the initial situation, like, e.g.,

$$\mathrm{On}(\mathrm{Block}_3, \mathrm{Table}, S_0) \wedge \mathrm{On}(\mathrm{Block}_3, \mathrm{Table}, S_0).$$

Now from the effect axiom together with the precondition axiom we cannot conclude anything about the location of blocks number two and three at time point $\mathrm{Move}(\mathrm{Go}(\mathrm{Block}_1, \mathrm{Table}), s))$: Are they still sitting on the table? Intuitively, the action of moving block number one should have no effect on the location of the other two blocks. Especially, we do not want to include the fact that their location is not affected by the action into the effect axiom — doing so in general would result in huge axiomatizations, if it is feasible at all. This problem of specifying what is, and what is not, affected by an action has become known as the frame problem — it has first been described by McCarthy and Hayes in 1969 [McCarthy and Hayes, 1969]. The name given to the problem derives from a physics expression — here the "frame of reference" denotes those parameters that remain unchanged.

[1] In this introductory example we gloss over some details of the model — e.g., we ignore the fact that Block_1 may be blocked. A fully worked model is featured in chapter 2.

The frame problem spawned a large number of solution attempts, and, over time, also solutions. It was one of the problems driving the research in non-monotonic logic — maybe even the most important. For example, McCarthy himself invented the formalism of circumscription with solving the frame problem being among his goals [McCarthy, 1980]. But eventually, a solution to the frame problem was also found in classical, monotonic logic by Reiter: A comprehensive presentation of this solution can be found in [Reiter, 2001a], the most up to date book on the Situation Calculus. The basic idea of Reiter's solution to the frame problem is to enumerate for every fluent all the actions that might affect it; in our small example domain this could look as follows:

$$\begin{aligned}\mathrm{On}(x, y, \mathrm{Do}(a, s)) \equiv\ & a = \mathrm{Move}(x, y) \vee \\ & (\mathrm{On}(x, y, s) \wedge \neg(\exists z) z \neq y \wedge a = \mathrm{Move}(x, z))\end{aligned}$$

This is to say that a mutable property of the domain continues to hold unless it explicitly is changed by some action.

The use of a logical equivalence in the axiom means that we assume that the right hand-side of the equivalence constitutes an explanation closure for the fluent: It enumerates all the possible reasons for the fluent to be true at some time point; there cannot be other causes. In some cases such a strict solution to the frame problem may be undesirable: We might want to allow the truth values of some fluents to freely fluctuate from one timepoint to another. Consider, e.g., a fluent that is not under the sole control of the agent. A solution to this issue, as well as a detailed treatment of Reiter's solution to the frame problem, is contained in chapter 2.

Event Calculus

The Event Calculus has been introduced in [Kowalski and Sergot, 1986] as a logic programming formalism. It has later been reformulated in logic, appealing to circumscription to solve the frame problem [Shanahan, 1995]. The major difference between the Situation and the Event Calculus is that the latter uses a linear time structure — the natural, or the positive real numbers. There are two book-length treatments of the Event Calculus: In [Shanahan, 1997] the reader can find material on both the logic programming and the circumscription version, whereas the more recent [Mueller, 2006] focuses on the circumscription version only.

Different versions of the Event Calculus have been applied to different problem types — for example, Event Calculus has been used to abduce which actions must have occurred given some observations. For our "agent" metaphor this kind of reasoning is not needed — the agent should always know which actions it has executed. In chapter 2 below we introduce a version of the Event Calculus that is suitable for

planning, and, at the same time, does not appeal to circumscription for solving the frame problem; it stays entirely within classical first order logic instead.

Fluent Calculus

The Fluent Calculus is nowadays usually presented as a modern extension of Reiter's Situation Calculus. The major difference to the latter has been the introduction of a formal notion of a state — this is a first order term describing collections of fluents. Originally introduced also as a logic programming formalism in [Hölldobler and Schneeberger, 1990], it initially also had strong roots in non-classical logic, namely linear logic. It has been reformulated in [Thielscher, 1999a] appealing to intuitions familiar from classical logic instead. The frame problem has been solved in the Fluent Calculus by applying first order quantification to state terms.

Another important difference between the Fluent and the Situation Calculus is that the former is action-centered while the latter is fluent-centered — at least in the popular version based on Reiter's successor state axioms. That is to say, in Fluent Calculus we specify for each *action* the effects it has while a successor state axiom specifies for a *fluent* by which actions it is affected. For our agent, who is chiefly interested in the effects of its actions, this action-centrism seems to be more natural. It has also been shown that this action-centered modeling has inferential advantages compared to the fluent-centered modeling [Thielscher, 1999b].

In the most recent reformulation [Thielscher, 2007] of the Fluent Calculus the formal notion of a state has been eliminated, so that the main difference between the Fluent and the Situation Calculus now is that between being action-based and being fluent-based. It turns out that, for this version of the Fluent Calculus, first order quantification over single fluents (instead of collections of fluents) is also enough to solve the frame problem. It is this most recent version of the Fluent Calculus that we are using in this thesis — a formal introduction can again be found in chapter 2.

1.1.2 Planning Languages

The first, and most famous, dedicated planning system that has been developed is STRIPS [Fikes and Nilsson, 1971]. Here, the initial state is specified by a set of (positive) fluents, and actions are identified with "add" and "delete" operations on this set. A plan consists of a sequence of actions achieving some goal — planning is the problem of finding a plan. Compared to the major action calculi discussed above this approach is of very restricted expressivity: We can only express complete information — every fluent is either true (contained in the state) or false (otherwise).

Even though there have been numerous expressive extensions to this very basic planning language these languages are still quite restrictive. Our action logic programs will be capable of finding solutions to planning problems formulated in

unrestricted action calculi.

1.2 Action Programming Languages

Let us now turn to the main topic of this thesis — action programming languages. These are programming languages that can be used to specify the strategy used by an agent in some dynamic domain. The program determines the sequence of actions that the agent is executing, trying to achieve its goal.

Already in the same year when the frame problem has been discovered, in [Green, 1969] the idea of solving planning problems in the Situation Calculus by resolution theorem proving has been put forth. Action logic programs are based upon the same idea: Theorem proving atop of logic-based action theories.

Up to now — to the best of our knowledge — only three action programming languages that use an action calculus world model in classical logic have been proposed:

- Golog, which is built atop of the Situation Calculus [Levesque et al., 1997, Reiter, 2001a];
- an unnamed robot control language which is based on the Event Calculus [Shanahan and Witkowski, 2000]; and
- Flux, which uses the Fluent Calculus as its foundation [Thielscher, 2005a, Thielscher, 2005d].

In the following we give a brief introduction to these three languages, before taking a high-level view at their merits and their shortcomings. Then we provide a first glimpse at the action logic programs presented in this thesis — these are meant to combine the nice features of the three aforementioned languages.

1.2.1 Golog

Golog provides Algol-inspired constructs for defining agent strategies. Thus in the basic version it features, e.g., `if-then-else` conditionals, `while` loops, and procedures. These programming constructs are read as shorthand for Situation Calculus formulas — i.e. they are macro-expanded to (sometimes second-order) Situation Calculus formulas. Later on, this second-order logical semantics has been replaced by a transition semantics: For this the programs themselves must also be encoded as terms in the resulting Situation Calculus theory.

1.2.2 Event Calculus-based Language

The unnamed robot control language from [Shanahan and Witkowski, 2000] is based on the idea to directly use a logic programming implementation of an Event Calculus-

based dynamic world model to control the robot. There does not seem to be a distinction between the world model, and the strategy employed. The language also refers to epistemic fluents for which no well-understood, logical counterpart has been developed in the Event Calculus, yet. This, as well as the usage of negation-as-failure in the logic program complicates the task of giving a declarative semantics to the language. To be fair the main focus of the work is to show how logic-based high-level reasoning can be done atop of low-level robot control.

1.2.3 Flux

Flux programs are full constraint logic programs. That is, there is an implementation (in ECLiPSe Prolog) of a certain fragment of the Fluent Calculus that can be used to model the dynamic domain. Moreover, an arbitrary constraint logic program can be used on top of this world model to specify the strategic behavior of an agent. Because Flux programs may contain arbitrary non-logical features of ECLiPSe Prolog their semantics has not been given as a logical semantics, but using the notion of a computation tree instead [Thielscher, 2005a].

1.2.4 Action Logic Programs

There is one feature that all three aforementioned languages share: They are tied to a specific action calculus that must be used to model the dynamic domains. (This claim has to be qualified: There has been substantial research on how to translate, e.g., Fluent into Situation Calculus.)

Moreover, none of these three languages fully separates the strategy an agent employs from the world model. Imagine we want to define some kind of measure on fluents that currently hold in order to guide our strategy — something like saying "if at least five pawns are located at...". Neither in Golog nor in the unnamed Event Calculus-based language can we add the necessary axioms to the underlying action theory. Hence, the formulas defining this measure have to be already there. In Flux such auxiliary formulas can be added to the underlying action domain in a natural way. However, in Flux the strategy programs contain state terms, and thus the separation between strategy and action domain is again not complete.

Finally, we observe that the three languages are more or less procedural, and thus none of them has been provided with a straightforward declarative semantics. Golog is based directly on procedural programming constructs, and has been provided with both a second-order logical semantics, and a transition semantics. The unrestricted logic programming paradigm employed by the other two languages has so far precluded a logical semantics.

With the action logic programs that we are about to present we address these shortcomings. In particular, action logic programs

- are largely independent from a particular action calculus (they use a more general action calculus that encompasses, e.g., the Event, Fluent, and Situation Calculus);
- draw a clear distinction between the world dynamics, and the strategy employed; and
- are a fully declarative logic programming paradigm that is based on the familiar first order semantics.[2]

Of the aforementioned three action programming languages the closest relative of action logic programs is Flux. Indeed, it is only fair to say that action logic programs are a fully general reconstruction of Flux in classical logic that is moreover independent from the Fluent Calculus.

1.3 Logic-based Knowledge Representation and Reasoning about Actions

Almost all the theoretical work in this thesis takes place in the context of classical first order logic. In particular, action logic programs will be a purely first order formalism. That is to say we will use first order logic to specify both the dynamic domain (in an action calculus), and the strategy that is used to achieve a goal in that dynamic domain.

However, first order logic is far from being the sole logic that has ever been proposed. Let us hence at this point make a digression: In this section we will briefly outline some of the available logical alternatives. In particular we will give pointers to our motivation for choosing first order logic as formalism underlying action logic programs.

Of course, we cannot give a comprehensive treatment of such a broad subject, and we are very much aware that such a cursory treatment will not be satisfactory. On the other hand, not mentioning the issue at all does not appear to be satisfactory, either. The reader looking for an overview of the multi-faceted use of logic in knowledge representation is referred to [Gabbay et al., 1998], or the more recent [Lifschitz et al., 2007].

1.3.1 First Order Logic

Historically, what is now known as classical first order logic ($\mathcal{FOL}$) originates with Frege's famous Begriffsschrift [Frege, 1879]. Frege's work marks the starting point

[2] Only theorem 3.3 rests on second order assumptions.

of modern logic, introducing the predicate calculus and thus overcoming the limitations of the up to then standard Aristotelian logic. Strictly speaking the predicate calculus Frege proposed in modern terms is second order — it contains variables that range over sets made up from elements of the underlying domain of interpretation. Restricting Frege's system by allowing variables only to range over single elements of the domain of interpretation we obtain basically what is now known as classical first order logic.

This idea of interpreting logical formulas is the second major innovation that sets modern first order logic apart from Aristotelian logic — this has led to the development of model theory (cf., e.g., [Chang and Keisler, 1990, Hodges, 1997]). This method was first used to prove that the parallel postulate in Euclidean geometry is independent from the other axioms, by giving two different interpretations of the remaining axioms. This idea of interpreting formulas allows an elegant formulation of both knowing a proposition ϕ as well as not knowing it. We know ϕ holds if it is true in every interpretation — and we do not know whether ϕ holds if there are interpretations that make ϕ true, as well as interpretations that make ϕ false.

Using this idea of interpretation, in first order logic predicates and terms are assigned to relations on, or elements from, some universe of discourse. The idea may be illustrated by Quine's dictum "To be is to be the value of a variable".

A natural question that arises here is: What is the domain of interpretation (the universe of discourse)? For first order logic the answer is any non-empty set. The underlying idea is that the notion of logical consequence should depend on logic alone and not on the structure of the underlying domain. This may be understood as saying logic has no (particular) ontology — a paraphrase of Putnam's dictum. Something follows logically if the reasoning does not depend on particular properties of the domain we are reasoning about. Or, as Hilbert put it: "One must be able to say tables, chairs, beer-mugs each time in place of points, lines, planes".

The domain being an arbitrary non-empty set leads to a second remark: Just what is any set? First order model theory is usually done in the context of Zermelo-Fraenkel set theory with the axiom of choice (ZFC), which in turn is a first order theory. How does this not constitute a vicious circle? The commonly agreed upon answer is that the notions of "set", but also "for all", "and" etc. that are used for defining the semantics of first order logic are pre-formal. We obviously cannot go on to provide formal foundations for formal theories forever. Hence formal first order model theory seems to rest on the assumption that ZFC correctly captures our intuitions about sets.

A different justification for $\mathcal{FOL}$ might be provided by Gödel's completeness theorem [Gödel, 1930]. This states that in first order logic the semantic, model-theoretic consequence relation coincides with the syntactic consequence relation, i.e. the consequence relation defined via proof. The proof-theoretic consequence relation that Gödel used in his proof of the completeness theorem is the modus ponens proof rule,

which arguably is very intuitive. By now also many other proof systems for first order logic have been shown to be complete. What Gödel's completeness theorem tells us is that to object against the set-based model-theoretical semantics of first order logic, is also to object against an arbitrary complete proof system for first order logic — and vice versa. So, if we have doubts about the semantic consequence relation, these may be eliminated if we trust the proofs.

1.3.2 Second Order Logic

In second order logic variables range not only over individual elements of the domain but also over functions and relations thereon, i.e. over sets. Set variables are thought to range over all subsets of the domain of interpretation, not only the first order definable ones — at least in the standard semantics. In the alternative, less commonly used Henkin semantics set variables range over first order definable sets only.

This feature makes standard second order logic extremely powerful. For example, it semantically captures the concept of "a smallest set such that": It is possible to enforce a certain cardinality of the domain of interpretation, i.e. second order logic is categorical. First order logic on the other lacks this power as witnessed by the Löwenheim-Skolem theorems.

This expressive power of second order logic comes at a price, however: Second order logic does not admit a sound and complete proof system under the standard semantics — this is a striking contrast to first order logic. This mismatch is due to the fact that second order logic lacks a property called compactness: This holds if a set of formulas (a theory) in a logic is satisfiable if and only if every finite subset thereof is satisfiable. Compactness ensures that for every theorem of a theory there is a finite subset of the axioms of the theory that the theorem can be derived from. Lindströms theorem [Ebbinghaus et al., 1994] assures us that first order logic is the strongest logic closed under classical negation that enjoys the compactness property. If we accept the tenet of classical proof theory that a proof is a finite object (that can be checked in finite time) this means that first order logic is the strongest classical logic that admits a complete proof system.

Still, attracted by the expressive power, researchers in knowledge representation have frequently resorted to second order formalisms. For example, second order logic has been used in the Situation Calculus to characterize semantically the tree of situations — the common, non-standard first order models are thus eliminated. Using a non-monotonic logic like circumscription to solve the frame problem also constitutes a recourse to second order features: Non-monotonic logics are usually based on the idea of minimizing the extension of certain predicates, and this idea of a smallest extension is a typical application of second order ideas. As a last example, in the Fluent Calculus a second order axiom has been used to ensure that in every

model of an axiomatization every possible combination of fluents is accounted for. The question whether using second order logic really is necessary will occasionally be resurfacing throughout this thesis; but by no means is this thesis an attempt to give an exhaustive treatment of this delicate subject.

A concise introduction to second order logic can be found in [Leivant, 1994]; additional material can also be found in [Andrews, 1986]. Currently probably the most comprehensive book on the philosophical questions surrounding second order logic is [Shapiro, 1991], a book that strongly advocates using second order logic for mathematics.

1.3.3 Intuitionistic Logic

Intuitionistic logic [van Dalen, 1994, Dummett, 1977] — which originates with the work of Brouwer [Brouwer, 1907] — constitutes the first modern alternative to classical predicate calculus. It deviates from classical logic in that it rejects the law of the excluded middle, i.e. intuitionistic logic rejects the idea that for any sentence Ψ it holds that $\Psi \vee \neg\Psi$ evaluates to true. Semantically, under the Brouwer-Heyting-Kolmogorov interpretation the meaning of a sentence is identified with a proof of the sentence. This idea can nicely be described by borrowing the slogan "The meaning of a sentence is the method of verification of the sentence" from the logical positivists of the Vienna Circle.

Without doubt intuitionistic logic constitutes a viable alternative to classical logic in many respects; however, from the perspective of logic-based knowledge representation the classical idea of truth in some structure (world model) in some respects is intuitively more appealing. This is particularly evident in the research area of reasoning about actions: For example, assume the agent's world model is given by a classical logical formula, describing the potentially many different worlds the agent deems possible. The effects of the agent's actions can then very naturally be described by updating (via set operations) each of the possible worlds individually [Winslett, 1990]. Expressing similar ideas in intuitionistic logic is not straightforward.

1.3.4 Sub-structural Logic

Sub-structural logic [Restall, 2000] has become the umbrella term for a number of logics that are neatly characterized by imposing restrictions on the structural rules in proof systems for classical logic like, e.g., natural deduction or the sequent calculus. The three most prominent members of this family are relevance logic, resource conscious logics and Lambek calculi/categorical grammars.

The latter play an important role in natural language processing in that they provide non-commutative versions of the logical connectives "and" and "or". The

basic idea can be illustrated by contrasting, e.g., the statements "He swallowed some pills and got ill" and "He got ill and swallowed some pills".

Relevance logic aims at overcoming perceived shortcomings of classical, material implication. For one in classical logic the principle "ex falso quodlibet" holds: Absolutely anything is implied by a contradiction and likewise a logical truth is implied by absolutely anything. Second, classical logic entails that for arbitrary sentences Φ and Ψ it holds that $(\Phi \supset \Psi) \vee (\Psi \supset \Phi)$. Statements like "If you like pancakes, then two plus two equals four" or "If elephants are grey then water is wet or if water is wet then elephants are grey" illustrate the kind of reasoning relevance logic tries to avoid. The aim is that implications should have only relevant premises. Relevance logic has also proven to be very useful for natural languages processing.

Resource Conscious Logics

Research in reasoning about actions has frequently used resource conscious logics. The most prominent representative of these logics is linear logic, introduced by Girard in [Girard, 1987]. It extends classical logic in such a way that it becomes possible to treat propositions as resources that are consumed by proofs. Treating propositions as resources is particularly interesting from the perspective of reasoning about actions and change because it constitutes a solution to the frame problem. A close relative of linear logic is (an extension of) Bibel's linear connection method, originally introduced in [Bibel, 1986]. The relationship between these formalisms as well as their application to planning problems is discussed in [Bibel, 1998].

The idea of treating propositions as resources also figures prominently in the early history of Fluent Calculus. The original version of Fluent Calculus introduced in [Hölldobler and Schneeberger, 1990] treats states as multi-sets: The state term Carries(Coin) $\circ$ Carries(Coin) expresses that there are two coins being carried. In Thielscher's version of the Fluent Calculus [Thielscher, 1999c], however, this term is provably equal to Carries(Coin) — the state-forming $\circ$ operator is idempotent, and in fact may be read as analogous to classical conjunction. The original non-idempotent $\circ$-operator in turn is best understood as analogous to linear conjunction.

Though attractive with regard to expressivity, sub-structural logics arguably suffer from an overly complicated model theory: They are typically defined by placing restrictions on proof theory.

The interested reader will find in the appendix of [Girard, 2001] a pronounced, if not polemic, articulation of the viewpoint that the model-theoretical semantics of $\mathcal{FOL}$ is not really necessary — and that sub-structural, proof-theoretically defined logics are fully adequate. For example, it is argued that we only need a model-theoretic semantics if we want to "interpret Taliban as students". The same opinionated appendix also features a number of criticisms of non-monotonic logics. We discuss these logics next.

1.3.5 Non-monotonic Logic

Classical logic is monotonic in the sense that if we add an axiom to a set of axioms the set of theorems never decreases. The family of non-monotonic logics [Brewka et al., 2008] differs in that by adding axioms we may lose theorems. The common idea underlying non-monotonic logics is to introduce some means to minimize the extension of certain predicates. Different notions of minimality give rise to different non-monotonic logics. Instead of treating all models of a formula equally in non-monotonic logics we prefer some (the minimal) models — hence often the expression of preferential semantics is used.

The development of non-monotonic logics was mainly motivated by two aims: For one, artificial intelligence researchers were looking for a way to formalize default reasoning with exceptions: As a famous example, we might want to express that typically birds fly. As a second motivation, researchers were hoping that non-monotonic logic would help them to overcome the frame problem. And as witnessed by, e.g., the circumscriptive Event Calculus they eventually succeeded.

In non-monotonic logic two issues arise: For one, the notion of model minimality makes it hard to express disjunctive information — the difficulty arises from trying to say at the same time that something might be true, and that, by default, we assume that everything is false unless we have conclusive evidence to the contrary. For this issue a number of solutions are available now. The second issue is that reasoning in non-monotonic logic is typically more expensive than reasoning in classical logic. This again is due to the preferential semantics, and the second order features involved. For example, for the full circumscriptive Event Calculus we do not even know in principle how to compute the logical consequences — all we have is a fragment for which reasoning can be reduced to first order reasoning.

1.3.6 Modal Logic

Modal logic has extensively been studied with the goal of formalizing modalities: Examples include possibility/necessity, obligations, or epistemics. A textbook introduction can, e.g., be found in [Blackburn et al., 2001]. The key advancement in modal logic in the last century was Kripke's formalization of their semantics via so-called possible worlds and accessibility relations. A world here may typically be regarded as a (consistent) set of propositional literals. Different modal logics are now definable by imposing different conditions on the accessibility relation between worlds. For example, we might say that we consider some proposition to be possible if there exists a possible world accessible from the current world where this proposition indeed is true.

From the perspective of reasoning about action these accessibility relations between worlds are clearly attractive: An action is simply taking us from the current

world state to the next world state. All we have to ensure is that the worlds accessible via some action accurately reflect the action's effects. Some early work in this direction has been conducted by Herzig and colleagues (cf., e.g. [Castilho et al., 1999]). Recently also a reformulation of the Situation Calculus in modal logic has appeared — here the worlds are not restricted to propositional logic but first order instead [Lakemeyer and Levesque, 2004, Lakemeyer and Levesque, 2005].

1.3.7 The Choice for First Order Logic

With this plethora of different logics at our hand, we have chosen to base our research on first order logic only. This choice was motivated by the following observations: First, none of the logics proposed so far "gets it right". That is, up to now there is no single logic that does not have some shortcoming (at least when applied to certain problems it was not intended for). Second, first order logic is conceptually very simple, yet it is very powerful. For this work we adopt the view that this conceptual simplicity of $\mathcal{FOL}$ trumps the (typically) more fine-grained expressivity of the competing logics. This argument may be viewed as taking an engineering perspective: Because $\mathcal{FOL}$ is conceptually simpler, it is easier to use. Of course, it has shortcomings. A naive axiomatization of a scenario where you try to shoot a turkey might not have the intended logical consequences — however, the same holds of many a non-monotonic logic. We conjecture that, for any logic, it is not hard to find natural language scenarios that — axiomatized naively — seem to imply that the logic fails to handle the scenario correctly. On the other hand, we argue that both the Tarskian semantics and the various $\mathcal{FOL}$ proof calculi are intuitive and simple enough, so that a knowledge engineer can employ $\mathcal{FOL}$ to correctly capture the scenario. A last argument that can be made for directly using first order logic is that it may be viewed as the "lingua franca" of computer science. This view has been expounded, e.g., in [Robinson, 2000].

For action logic programs we take first order logic only, and see how far we can push it. We will see that the proof of theorem 3.3 hinges on the presence of a second order axiom. We will also see that for some classes of action domains a form of non-monotonic reasoning may be called for (cf. the discussion at the very end of section 6.2). But overall action logic programs will show that for most aspects of modeling the behavior of single agents in dynamic domains classical first order logic is enough.

1.4 Structure of the Thesis

The rest of this thesis is organized as follows:

- In chapter 2 we formally introduce (constraint) logic programs, action calculi,

and Description Logics — material that action logic programs are built atop.

- In chapter 3 we introduce the action logic program framework — the main theoretical contribution of this thesis.
- The chapters 4 and 5 are devoted to presenting our implementation of two fragments the theoretical framework presented in chapter 3. We will present one implementation of open world reasoning over essentially propositional domains, and one implementation that is based on the knowledge representation formalism of Description Logics.
- In chapter 6 we explore the relation of our framework to existing work — it is here where we will further elaborate on the relation of action logic programs to existing languages like Golog and Flux.
- Finally, in chapter 7 we conclude.

2 Preliminaries

In this chapter we introduce the formal preliminaries of our work. It contains an introduction to

- the notation used, and pointers to the literature on first order logic in section 2.1;
- logic programming in section 2.2;
- Thielscher's unifying action calculus in section 2.3; and
- Description Logics in section 2.4.

2.1 First Order Logic and Notation

Action Logic Programs are based on many-sorted, classical first order logic with equality interpreted as identity. We assume the reader is already familiar with it. A concise summary of the most important results can be found in [Davis, 1993] while [van Dalen, 1994, Enderton, 1972, Ebbinghaus et al., 1994, Shoenfield, 1967] give more extensive treatments. We will abbreviate classical first order logic with equality by the acronym $\mathcal{FOL}$.

On Notation: The logical connectives are $\wedge, \neg, \forall$ with $\supset, \vee, \equiv, \exists$ treated as the usual abbreviations. Equality is written as $=$ in infix notation as is the common abbreviation $\neq$. Occasionally we will use a version of first order logic with counting quantifiers; these are written as $\geq n$ and $\leq n$, respectively, with $n \in \mathbb{N}$, like for example in $(\geq 1\ x)P(x)$. First order logic with equality and counting quantifiers is abbreviated $\mathcal{FOL}_{\leq,\geq}$, while $\mathcal{FOL}^2$ denotes first order logic with two variables. Capital letters are used for predicate and function symbols, while lower case letters denote variables. Greek letters, e.g., ϕ, ψ, denote formulas while atoms and literals are written as, e.g., $P(A), \neg P(B)$. Occasionally we let Latin letters, e.g., A, B, H range over atoms as we will use L for literals — this usage will be clear from the context. Sequences of variables and (more general) terms are written as $\vec{x}$ and $\vec{t}$, respectively. A formula with free variables $\vec{x}$ may be written as $\phi(\vec{x})$, and the universal closure of a formula ϕ is denoted as $(\forall)\phi$. In this thesis formulas with free variables are to be read as their universal closure unless explicitly stated otherwise. By $(\exists!x)\phi(x)$ we abbreviate the $\mathcal{FOL}$ formula expressing that there is a unique x

such that $\phi(x)$. The symbols $\top$ and $\bot$ stand for (arbitrary) tautologies and contradictions, respectively. Sans serif letters, e.g., A, range over sets of formulas. Since this is established practice we will denote substitutions by Greek letters, too — see section 2.2.1 below. Sets of numbers are denoted by the usual $\mathbb{N}, \mathbb{R}$ etc.. As usual by $\models$ we denote model-theoretic consequence while $\vdash$ denotes provability. Calligraphic letters are used for denoting both logics and logical languages. For example, $\mathcal{ALC}$ denotes a basic description logic and $\mathcal{L}$ may be used for a particular logical language. Note that we do not apply these notational conventions to description logics, but follow the notational conventions of the DL community instead.

2.2 Logic Programming

Logic programming comes in many different flavors. In this thesis only definite logic programs and their extension to constraint logic programs play a role. For both we recall the theoretical basics below. Note that the logic programming formalisms that we introduce below are based on the classical first order semantics — though many formalisms with a more specialized semantics have also been called logic programming.

2.2.1 Definite Logic Programs

Definite logic programs correspond to the Horn fragment of first order logic together with SLD-resolution. They are computationally complete and form the logically pure core of the programming language Prolog. The reader looking for an more in depth introduction is referred to the classic [J.W. Lloyd, 1987] or the more recent [Apt, 1996]. We assume the reader also knows the basics programming in Prolog — cf., e.g., [Clocksin and Mellish, 1987] and [O'Keefe, 1990] in addition to the above references.

Horn Clauses and Substitutions

A first order *Horn clause* is a universally quantified disjunction of first order literals $(\forall)L_1 \vee \ldots \vee L_n$, of which at most one is positive. Both *definite clauses* and *goal clauses* are Horn clauses: A definite clause π contains a positive literal whereas a goal clause γ does not. A definite clause $(\forall)H \vee \neg B_1 \vee \neg B_n$, where H is the (only) positive literal, may equivalently be written as an implication $(\forall)H \subset B_1 \wedge \ldots \wedge B_n$. It is common to refer to H as the head atom and to the B_i as the body atoms. A definite logic program P is a finite set of definite clauses. The declarative semantics of a program is given by its first order semantics. A query $(\exists)\varrho$ is an existentially quantified conjunction of first order atoms.

A *substitution* θ is a finite set of pairs $\{(x_1/t_1), \ldots, (x_n/t_n)\}$, where all the x_i are pairwise distinct variables and each t_i is a term different from x_i. Given a definite clause π and a substitution θ, by $\pi\theta$ we denote the definite clause obtained by substituting every occurrence of variable x_i in π by the term t_i simultaneously, for $\{(x_i, t_i) \in \theta \mid 1 \leq i \leq n\}$. Likewise, for a term t, and a substitution $\theta = \{(x_1/t_1), \ldots, (x_n/t_n)\}$ by $t\theta$ we denote the term obtained by simultaneously replacing every occurrence of a variable x_i in t by t_i. The empty substitution is denoted ϵ.

The composition $\theta_1\theta_2$ of two substitutions $\theta_1 = \{(x_1, t_1), \ldots, (x_n, t_n)\}$ and $\theta_2 = \{(y_1, s_1), \ldots, (y_m, s_m)\}$ denotes the substitution that is obtained from the substitution $\{(x_1, t_1\theta_2), \ldots, (x_n, t_n\theta_2)\} \cup \theta_2$ by deleting any pair $(x_i, t_i\theta_2)$ with $x_i = t_i\theta_2$ and any pair y_j, s_j for which $y_j \in \{x_1, \ldots, x_n\}$. A substitution θ_1 is more general than a substitution θ_2 if there is a substitution θ_3 such that $\theta_1\theta_3 = \theta_2$. A substitution θ is a unifier of two literals L_1, L_2 if $L_1\theta$ and $L_2\theta$ are syntactically identical. A clause π' is a fresh variant of a clause π if it is obtained by uniformly substituting each variable in π by an hitherto unmentioned variable.

SLD Resolution

The proof calculus SLD-resolution [Hill, 1974, van Emden and Kowalski, 1976] provides the operational counterpart to the declarative first order semantics of definite logic programs P.

SLD-resolution is a negative proof calculus. Thus, given a program P and a query $(\exists)\varrho$, it is established that $\mathsf{P} \models (\exists)\varrho$ by proving unsatisfiability of $\mathsf{P} \cup \{\neg(\exists)\varrho\}$. Negating a query $(\exists)\varrho = (\exists)G_1 \wedge \ldots \wedge G_n$ we obtain the goal clause $\gamma = (\forall)\neg G_1 \vee \ldots \vee \neg G_n$. A *state* is a pair $< \gamma, \theta >$, where γ is a goal clause, and θ is a substitution. A *derivation* of a query $(\exists)\varrho$ is a sequence of states, starting with state $< \gamma, \epsilon >$, where ϵ is the empty substitution. A derivation is successful if it ends in $< \bot, \theta >$, where $\bot$ denotes the empty clause, i.e. a contradiction. A successful derivation is also called refutation. The unifier θ in the last state of a refutation is also referred to as computed answer substitution. A derived state to which the rule of inference presented below cannot be applied indicates a failed derivation.

SLD-resolution is defined by the following rule of inference:

$$\frac{< (\neg G_1 \vee \ldots \vee \neg G_i \vee \ldots \vee \neg G_n),\ \theta_1 >}{< (\neg G_1 \vee \ldots \vee \neg B_1 \vee \ldots \vee \neg B_m \vee \ldots \vee \neg G_n)\theta_2,\ \theta_1\theta_2 >}$$

where G_i is an atom and $(H \subset B_1 \wedge \ldots \wedge B_m)$ is a fresh variant of a clause in P such that G_i and H unify with most general unifier θ_2.[1]

We restate two well-known results from the literature, establishing soundness and completeness of SLD-resolution wrt. the declarative semantics.

[1] For definite clauses a unique most general unifier always exists.

Theorem 2.1 (Soundness of SLD-resolution)**.** *Let* P *be a definite logic program,* $(\exists)\varrho$ *be a query, and* γ *the corresponding goal clause. If there exists a SLD-derivation starting from* $< \gamma, \epsilon >$ *and ending in* $< \bot, \theta >$ *then* $\mathsf{P} \vDash (\forall)\varrho\theta$*.*

The completeness result below uses the following notion of a computation rule:

Definition 2.1 (Computation Rule)**.** *A* computation rule *is a function selecting a literal from a Horn clause to continue the derivation with.*

Theorem 2.2 (Completeness of SLD-resolution)**.** *Let as before* P *be a definite logic program and* γ *be the negation of the query* $(\exists)\varrho$*. If* $\mathsf{P} \vDash \gamma\theta_1$*, then there exists a successful SLD-derivation via any computation rule starting with* $< \gamma, \epsilon >$ *and ending in* $< \bot, \theta_2 >$*. Furthermore, there is a substitution* θ_3*, such that* $\theta_1 = \theta_2\theta_3$*.*

2.2.2 Constraint Logic Programming

Constraint logic programs are an extension of definite logic programs. They make available specialized algorithms for the evaluation of some goals, the so called constraints. For example there might be linear constraints in a constraint logic program, that are evaluated using, e.g., the simplex method.

More precisely, constraint logic programming — CLP(X) — is a scheme, that is to be instantiated with a suitable first order constraint domain axiomatization X[Jaffar and Lassez, 1987]. In addition to the ordinary atoms of plain logic programming in CLP(X) there are special atoms — the constraints — that are evaluated against the background constraint theory X. Below we recall the presentation of constraint logic programming that can be found in [Frühwirth and Abdennadher, 2003].

In CLP(X) states are pairs $< \bigvee_{i=1..n} \neg G_i, \sigma >$, where the constraint store σ is a first order formula. Substitutions are not applied to the goals, but collected as equality formulas in the constraint store instead.

The proof calculus of CLP(X) consists of the following two rules of inference, one for the program atoms, and one for the constraints:

- *Program Atoms*:

$$\frac{< (\neg G_1 \vee \ldots \vee \neg G_i \vee \ldots \vee \neg G_n),\ \sigma >}{< (\neg G_1 \vee \ldots \vee \bigvee_{i=1..m} \neg B_i \vee \ldots \vee \neg G_n),\ \sigma' >}$$

  where G_i is a program atom and $(H \subset B_1 \wedge \ldots \wedge B_m)$ is a fresh variant of any clause in the program P such that $\mathsf{X} \vDash (\exists)G_i = H \wedge \sigma$.[2] Furthermore the resulting constraint store σ' is set to $\sigma \wedge G_i = H$.

[2]The equation $G_i = H$ abbreviates equating the respective arguments.

- *Constraint Atoms*:

$$\frac{< (\neg G_1 \vee \ldots \vee \neg C \vee \ldots \vee \neg G_n),\ \sigma >}{< (\neg G_1 \vee \ldots \vee \neg G_n),\ \sigma' >}$$

  where C is a constraint atom and $\mathsf{X} \vDash (\forall) C \wedge \sigma \equiv \sigma'$.

For this proof calculus the following soundness and completeness results hold for program P, constraint theory X, query ϱ and constraint stores $\sigma_{(i)}$:

Theorem 2.3 (Soundness of CLP(X))**.** *If ϱ has a successful derivation with computed answer σ then $\mathsf{P} \cup \mathsf{X} \vDash (\forall)\sigma \supset \varrho$.*

Theorem 2.4 (Completeness of CLP(X))**.** *If $\mathsf{P} \cup \mathsf{X} \vDash (\forall)\sigma \supset \varrho$ and σ is satisfiable wrt. X, then there are successful derivations for the goal G with computed answers $\sigma_1, \ldots, \sigma_n$ such that $\mathsf{X} \vDash (\forall)(\sigma \supset (\sigma_1 \vee \ldots \vee \sigma_n))$.*

Observe that the answers in CLP(X) are conditional: for an answer $\bigvee_{i=1..k} \sigma_i$ to a query we have to check whether $\mathsf{X} \vDash \bigvee_{i=1..k} \sigma_i$ to see whether the query is indeed entailed.

The proof calculus for CLP(X) provides the operational semantics of constraint logic programs; their declarative semantics is given by the usual first order semantics. In the classic work on the semantics of constraint logic programs [Jaffar et al., 1998] these two semantics are complemented by an algebraic semantics. For this one specifies an algebra as the domain of computation for X such that the evaluation of constraints can be done by algebraic operations rather than by logical proofs on the constraint theory. For our purposes, introducing the logical and operational semantics suffices.

2.3 Unifying Action Calculus

In this section we recall the essentials of a recently proposed unifying action calculus (UAC) [Thielscher, 2007]. The unifying action calculus has been introduced with the stated goal of bundling research efforts in the reasoning about action community. It has been shown that this calculus abstracts from, and can be instantiated by, concrete calculi such as the Event, Fluent, or Situation Calculus. It can also be instantiated by less expressive but computationally more feasible planning languages such as STRIPS and ADL. This section contains excerpts from [Thielscher, 2007].

2.3.1 Formal Basics

Formally, the UAC is based on many-sorted first order logic with equality and the four sorts TIME, FLUENT, OBJECT, and ACTION.[3] Fluents are reified, and the predicate Holds : FLUENT × TIME is used to indicate whether a particular fluent evaluates to true at a particular time. For axiomatizing action preconditions the predicate Poss : ACTION × TIME × TIME is used. Usually in action calculi the predicate Poss has only one argument of sort TIME. UAC's Poss can be read as "There is a possible transition from one timepoint via an action to another timepoint". Having two arguments of sort TIME also allows modeling actions with indirect effects or varying duration. There are only finitely many function symbols into sorts FLUENT and ACTION, respectively. For actions and fluents unique name axioms are included by default.

The UAC abstracts from a particular time structure. Each domain axiomatization includes an axiomatization of the chosen underlying time structure. For example, the natural or the real numbers provide the linear time structure of the Event Calculus, while both the Fluent and Situation Calculus are based on the branching time structure of situations. It is mandatory that the time structure includes a constant denoting the earliest time point. Furthermore it is assumed that it defines a possibly partial order on the time structure, denoted by the relation $s_1 < s_2$.

2.3.2 Domain Axiomatizations

The following axiomatization of the blocks world illustrates the basic building blocks of UAC's action domain axiomatizations. In the well known blocks world a number of blocks situated on a table can be moved around unless there are other blocks stacked on top. The goal is to arrange the blocks according to some pattern. The axiomatization is built on top of the branching time structure of situations:

Example 2.1 (Blocks World Axiomatization). We can move block_1 from one location x to another location y if there is neither another block block_2 on top of it, nor a block block_3 blocking location y. This *precondition* of moving a block is expressed by the following axiom:

$$
\begin{aligned}
(\forall)\text{Poss}(\text{Move}(\text{block}_1, x, y), s_1, s_2) \equiv \\
\quad \text{Holds}(\text{On}(\text{block}_1, x), s_1) \wedge x \neq y \wedge \\
\quad \neg(\exists \text{block}_2)\text{Holds}(\text{On}(\text{block}_2, \text{block}_1), s_1) \wedge \\
\quad (\neg(\exists \text{block}_3)\text{Holds}(\text{On}(\text{block}_3, y), s_1) \vee y = \text{Table}) \wedge \\
\quad s_2 = \text{Do}(\text{Move}(\text{block}_1, x, y), s_1).
\end{aligned}
$$

[3] By convention variable symbols s, f, x, and a are used for terms of sort TIME, FLUENT, OBJECT, and ACTION, respectively.

If we move a block from location x to location y then the only changes that occur are, respectively, that the block is no longer located at x, but at y. These *effects* of moving a block are axiomatized as follows:

$$(\forall)\text{Poss}(\text{Move}(\text{block}, x, y), s_1, s_2) \supset$$
$$[(\forall f)(f = \text{On}(\text{block}, y) \vee (\text{Holds}(f, s_1) \wedge f \neq \text{On}(\text{block}, x))) \equiv \text{Holds}(f, s_2)].$$

Please note how first order quantification is used in the above axiom to solve the frame problem.

The following *domain constraint* expresses the fact that every block is situated at exactly one location at any given time:

$$(\forall s \exists! y)\text{Holds}(\text{On}(\text{block}, y), s).$$

Finally, suppose that the following axiom describes what is known about the *initial situation*:

$$(\forall f)\text{Holds}(f, S_0) \equiv f = \text{On}(\text{Block}_1, \text{Table}) \vee f = \text{On}(\text{Block}_2, \text{Table}).$$

Together with the unique name axioms for the blocks and the table this axiomatization entails

$$\text{Holds}(\text{On}(\text{Block}_2, \text{Block}_1), \text{Do}(\text{Move}(\text{Block}_2, \text{Table}, \text{Block}_1), S_0)).$$

Formally the different axiom types that are used to formalize a dynamic domain in the UAC are defined as follows:

Definition 2.2 (UAC Basic Formulas)**.** *Let $\vec{s}$ be a sequence of variables of sort* TIME *and A be a function into sort* ACTION*.*

- *A* state formula $\phi[\vec{s}]$ in $\vec{s}$ *is a first-order formula with free variables $\vec{s}$ where*
  - *for each occurrence of $\text{Holds}(f, s)$ we have $s \in \vec{s}$;*
  - *predicate Poss does not occur.*

  We say a state formula is pure *if Holds and equality are the only predicate symbol used.*

- *A* state property ϕ *is an expression defined as a first order formula without any occurrence of Poss or Holds, but allowing terms $F(\vec{x})$ of sort* FLUENT *to take the role of atoms. By $\text{HOLDS}(\phi, s)$ we denote the state formula obtained from ϕ by replacing every occurrence of a fluent f by $\text{Holds}(f, s)$.*

- *A* domain constraint *is a state formula in s:*

$$(\forall s)\delta[s].$$

- *A* precondition axiom *is of the form*

$$(\forall) Poss(A(\vec{x}), s_1, s_2) \equiv \pi_A[s_1],$$

where $\pi_A[s_1]$ *is a state formula in* s_1 *with free variables among* $s_1, s_2, \vec{x}$.

- *An* effect axiom *is of the form*

$$Poss(A(\vec{x}), s_1, s_2) \supset \eta_1[s_1, s_2] \vee \ldots \vee \eta_k[s_1, s_2].$$

The $\eta_i[s_1, s_2]$ *denote different cases; each of these sub-formulas is of the form*

$$\begin{array}{ll} (\exists \vec{y}_i)(\phi_i[s_1] & \wedge\ (\forall f)[\delta_i^+[s_1, s_2] \supset Holds(f, s_2)] \\ & \wedge\ (\forall f)[\delta_i^-[s_1, s_2] \supset \neg Holds(f, s_2)]) \end{array}$$

where $\phi_i[s_1]$ *is a state formula in* s_1 *with free variables among* $s_1, \vec{x}, \vec{y}_i$,[4] *and both* $\delta_i^+[s_1, s_2]$ *and* $\delta_i^-[s_1, s_2]$ *are state formulas in* s_1, s_2 *with free variables among* $f, s_1, s_2, \vec{x}, \vec{y}_i$.

- *An* initial state axiom *is a state formula in the least element of sort* TIME.

Before we can turn to the definition of an action domain axiomatization we still need to define what the foundational axioms of the UAC are:

Definition 2.3 (Foundational Axioms)**.** Foundational axioms $\mathsf{D}_{\mathsf{aux}}$ *contain an axiomatization of the (possibly partially) ordered underlying time structure that must feature a constant denoting the earliest time point. By default, unique name axioms for fluents and actions are also included. In addition* $\mathsf{D}_{\mathsf{aux}}$ *may also contain domain-dependent additional axioms, e.g., an axiomatization of arithmetic or finite domain constraints.*

Now we can define action domain axiomatizations in the UAC as follows:

Definition 2.4 (Domain Axiomatizations)**.** *An* action domain axiomatization D *consists of the sets* $\mathsf{D}_{\mathsf{Poss}}$ *and* $\mathsf{D}_{\mathsf{Effects}}$ *of precondition and effect axioms, respectively, each containing one axiom for every action, along with a finite set of domain constraints* D_{dc}*, a finite set of initial state axioms* $\mathsf{D}_{\mathsf{Init}}$ *and foundational axioms* $\mathsf{D}_{\mathsf{aux}}$.

2.3.3 Concrete Action Calculi in UAC

In this section we recall how specific action calculi are represented in the unifying action calculus. In particular this is done for the big three action calculi, the Event, Fluent, and Situation Calculus. We will not present such definitions for planning languages, since these usually can be represented in one of the big three action calculi.

[4] The purpose of sub-formula $\phi_i[s_1]$ is to define possible restrictions for case i to apply.

Fluent Calculus

We start by recalling how Fluent Calculus domains are represented in the unifying action calculus.

Definition 2.5 (Fluent Calculus Domain)**.** *A Fluent Calculus domain axiomatization in the UAC is subject to the following conditions:*

- *Every precondition axiom is of the form*

$$(\forall) Poss(A(\vec{x}), s_1, s_2) \equiv \pi_A[s_1] \wedge s_2 = Do(A(\vec{x}), s_1),$$

  where $\pi_A[s_1]$ is a state formula in s_1 with free variables among $s_1, \vec{x}$; and

- *Every effect axiom is of the form*

$$\begin{aligned} &Poss(A(\vec{x}), s_1, s_2) \supset \\ &\quad (\bigvee_k)(\exists \vec{y}_k)(\Phi_k[s_1] \wedge \\ &\qquad (\forall f)[(\bigvee_i f = f_{ki} \vee (Holds(f, s_1) \wedge \bigwedge_j f \neq g_{kj})) \supset Holds(f, s_2)] \wedge \\ &\qquad (\forall f)[(\bigvee_j f = g_{kj} \vee (\neg Holds(f, s_1) \wedge \bigwedge_i f \neq f_{ki})) \supset \neg Holds(f, s_2)]) \end{aligned}$$

  where the f_{ki} and g_{kj} are fluent terms with variables among $\vec{x}, \vec{y}_k$ — the positive and negative effect, respectively —, and the Φ_k are state formulas in s_1 with free variables among $s, \vec{x}, \vec{y}$. Positive and negative action effects in an effect axiom with k cases are subject to a natural consistency condition, namely, we require that

$$(\forall) \bigwedge_i \bigwedge_j f_{ki} \neq g_{kj}$$

  holds for all k. Effect axioms can equivalently be written as

$$\begin{aligned} &Poss(A(\vec{x}), s_1, s_2) \supset \\ &\quad \bigvee_k (\exists \vec{y}_k)(\Phi_k[s_1] \wedge (\forall f)[(\bigvee_i f = f_{ki} \vee (Holds(f, s_1) \wedge \bigwedge_j f \neq g_{kj})) \\ &\qquad\qquad \equiv Holds(f, s_2)]). \end{aligned}$$

- *Foundational axioms $\mathsf{D}_{\mathsf{aux}}$ contain an axiomatization of situations (the underlying time structure).*

The initial state axiom and the domain constraints are not subject to any special conditions.

Assume an action $A(\vec{x})$ is applicable in a situation s_1. The foundational theorem of the Fluent Calculus [Thielscher, 2005d] then establishes that a fluent holds in the situation $\mathrm{Do}(A(\vec{x}), s_1)$ if and only if

- it held at situation s_1 and was not a negative effect of A, or
- it is a positive effect of A,

given that a single state update equation applies. The foundational theorem shows that the Fluent Calculus solves the frame problem. The following theorem is the analogous result for Fluent Calculus domains in the unifying action calculus:

Theorem 2.5 (Fluent Calculus Foundational Theorem)**.** *Assume given a Fluent Calculus domain* D*, a Fluent Calculus effect axiom and a model* $\mathfrak{I}$ *for* D *such that*

- $\models_{\mathfrak{I}} \mathit{Poss}(A(\vec{x}), s_1, s_2)$*, and*
- $\models_{\mathfrak{I}} (\exists \vec{y}_k)\Phi_k[s_1]$*, for some* k*.*

Then

$$\models_{\mathfrak{I}} (\forall f)[\bigvee_i f = f_{ki} \vee \mathit{Holds}(f, s_1) \wedge \bigwedge_j f \neq g_{kj} \equiv \mathit{Holds}(f, s_2)].$$

Situation Calculus

For solving the frame problem in the Situation Calculus using classical first order logic so-called successor state axioms have been introduced in [Reiter, 1991]. The general form of these axioms, one for each fluent $F(\vec{u})$, is

$$\mathrm{Poss}(a, s) \supset [\mathrm{Holds}(F(\vec{u}), \mathrm{Do}(a, s)) \equiv \Gamma^+_F[s] \vee \mathrm{Holds}(F(\vec{u}), s) \wedge \neg\Gamma^-_F[s]]$$

where Γ^+_F and Γ^-_F describe the conditions on $a, s, \vec{u}$ under which fluent $F(\vec{u})$ is, respectively, a positive or a negative effect. Reiter's successor state axioms do not allow to model non-deterministic actions; in [Thielscher, 2007] they have been generalized to non-deterministic effects.

Precondition axioms in the Situation Calculus are defined exactly as in the Fluent Calculus (cf. definition 2.5). Standard deterministic successor state axioms in turn can be represented in the UAC as

$$\mathrm{Poss}(a, s_1, s_2) \supset [\mathrm{Holds}(F(\vec{u}), s_2) \equiv \Gamma^+_F[s_1] \vee \mathrm{Holds}(F(\vec{u}), s) \wedge \neg\Gamma^-_F[s_1]](*).$$

This form does not comply with the definition of an effect axiom (cf. definition 2.2) in the unifying action calculus, though. But it has been shown how both deterministic and non-deterministic successor state axioms can equivalently be compiled to effect

axioms in the unifying action calculus in [Thielscher, 2007], furthering the work begun in [Schiffel and Thielscher, 2006].

Thus we can safely regard successor state axioms of the form $(*)$ as shorthand for effect axioms. Assuming that the successor state axioms of a Situation Calculus domain axiomatization are represented as UAC effect axioms, too, will especially prove to be convenient when we later on consider properties of domain axiomatizations in general. But, strictly speaking, it is not necessary: we could likewise have loosened the definition of an effect axiom in the UAC, at the cost of a case distinction, complicating the presentation.

Event Calculus

The Event Calculus is the major proponent of an action calculus based upon linear time. It has originally been introduced as a logic program by Kowalski and Sergot in [Kowalski and Sergot, 1986]. It has later been reformulated in classical predicate logic together with circumscription by Shanahan; the classic book on the Event Calculus now is his [Shanahan, 1997]. A contemporary book on the many different aspects and uses of the Event Calculus is [Mueller, 2006].

A version of the Event Calculus that is well suited for solving planning problems has been presented in [Shanahan, 2000]; the robot programming language presented in [Shanahan and Witkowski, 2000] is based on this version, too. In this version the natural numbers $\mathbb{N}$ serve as the underlying linear time structure. This version of the Event Calculus is quite restrictive: e.g., one cannot ask whether there exists a plan achieving a goal at some time point; one can only ask whether the goal can be achieved at a specific time point.

Below we formally introduce our own variant of the Event Calculus for planning from [Shanahan, 2000]; we then show how it can be accommodated in the unifying action calculus. Our calculus is more general than the one from [Shanahan, 2000] in that it allows to ask whether there exists a plan achieving a goal at some time point; and in that it allows more general conditions in effect axioms, etc.: We allow first order fluent formulas instead of only ground formulas. Note that our version of the Event Calculus is also a restriction of the Event Calculus from [Shanahan, 2000], because we only allow non-concurrent action domains: Non-concurrency of the action domains is one of the intuitions that will be underlying Action Logic Programs.

Basics of Event Calculus In Shanahan's Event Calculus the frame problem is solved using a combination of foundational axioms and circumscription. Circumscription is the oldest non-monotonic logic, originally introduced by McCarthy in [McCarthy, 1980]. As such it allows to minimize the extension of a predicate — this operation of minimization is called circumscribing a predicate. Circumscribing predicates $P_1, \ldots, P_n$ wrt. a formula ψ is denoted $\mathrm{CIRC}(\psi; P_1, \ldots, P_n)$. We do not

need the full definitions of circumscription here; for the interested reader the definite treatment of circumscription to date is [Lifschitz, 1994].

Definition 2.6 (Event Calculus Foundational Axioms)**.** *In order to solve the fundamental frame problem the Event Calculus features the foundational axioms* EC *given below:* [5]

$$
\begin{aligned}
&\mathit{Holds}(f, s) \subset \mathit{Init}_P(f) \wedge \neg \mathit{Clipped}(0, f, s)\\
&\mathit{Holds}(f, s_2) \subset\\
&\quad \mathit{Happens}(a, s_1) \wedge \mathit{Initiates}(a, f, s_1) \wedge s_1 < s_2 \wedge \neg \mathit{Clipped}(s_1, f, s_2)\\
&\mathit{Clipped}(s_1, f, s_3) \equiv\\
&\quad (\exists a, s_2)[\mathit{Happens}(a, s_2) \wedge s_1 \leq s_2 \wedge s_2 < s_3 \wedge\\
&\qquad [\mathit{Terminates}(a, f, s_2) \vee \mathit{Releases}(a, f, s_2)]]\\
&\neg \mathit{Holds}(f, s_2) \subset\\
&\quad \mathit{Happens}(a, s_1) \wedge \mathit{Terminates}(a, f, s_1) \wedge s_1 < s_2 \wedge \neg \mathit{Declipped}(s_1, f, s_2)\\
&\neg \mathit{Holds}(f, s) \subset \mathit{Init}_N(f) \wedge \neg \mathit{Declipped}(0, f, s)\\
&\mathit{Declipped}(s_1, f, s_3) \equiv\\
&\quad (\exists a, s_2)[\mathit{Happens}(a, s_2) \wedge s_1 \leq s_2 \wedge s_2 < s_3 \wedge\\
&\qquad [\mathit{Initiates}(a, f, s_2) \vee \mathit{Releases}(a, f, s_2)]]
\end{aligned}
$$

Put in words, a fluent f holds if it held initially and has not been changed since; or if it has been initiated by some event (and not been changed since). A fluent that holds can be changed by an event that terminates it (sets it to false) or by an event that releases it from the frame assumption. The axioms that specify when a fluent does not hold are symmetric.

Definition 2.7 (Event Calculus Basic Formula Types)**.** *For the specification of concrete action domains the Event Calculus features a number of formula types. In our version of the Event Calculus we include the following:*

- *Conditions ϕ are used by all the basic formula types: they are first order formulas that contain only $\mathit{Holds}(f, s)$ literals, for fluent f and time point s. Moreover the time point variable s is the same in each literal (the condition is uniform in s). They can be seen as a restricted form of the state formulas of the UAC.*

- *Positive and negative effect axioms are used to specify the positive and negative effects of events; they are of the following form:*

$$
\begin{aligned}
&(\forall s)\phi(s) \supset \mathit{Initiates}(a, f, s), \text{ and}\\
&(\forall s)\phi(s) \supset \mathit{Terminates}(a, f, s),
\end{aligned}
$$

[5] We denote events by terms a, as in action.

for ground event a, ground fluent f, and time point variable s. Observe that events in the Event Calculus are instantaneous: the effects are initiated (terminated) immediately. Only by the foundational axioms EC *do the effects hold (not hold) at the next time point.*

- *Release axioms are used to specify those fluents that are exempt from the frame assumption due to an event occurrence. They are of the form:*

$$(\forall s)\phi(s) \supset \mathit{Releases}(a, f, s),$$

for ground event a, ground fluent f, and time point variable s.

- *State constraints are conditions that have to be fulfilled at every timepoint. Thus they are of the form*

$$(\forall s)\phi(s)$$

Planning Domains in the Event Calculus With the help of these basic formula types we can now define what an Event Calculus planning domain looks like:

Definition 2.8 (Event Calculus Planning Domains)**.** *A domain axiomatization for expressing planning tasks is specified in the Event Calculus with the help of the following formulas:*

- *The initial situation ϕ_0 is a Boolean combination of atoms of the form $\mathit{Init}_P(f)$ or $\mathit{Init}_N(f)$, for ground fluent f, meaning that initially the fluent does hold (P - positive) or does not hold (N - negative).*

- *The planning goal γ is of the form $(\exists s)\phi(s)$, where is ϕ is a Boolean combination of $\mathit{Holds}(f, s)$ literals, for ground fluent f and time point variable s, uniform in s.*

- *A totally ordered narrative δ is a conjunction of ground atoms of the form $\mathit{Happens}(a, s)$, for event a and time point s, such that every s occurs exactly once in the conjunction. Intuitively, a narrative constitutes a plan.*

- *A domain description $\mathsf{D_{EC}}$ contains a finite number of positive and negative effect axioms, and release axioms. Additionally, a conjunction of state constraints might be included. Finally, unique name axioms for fluents, actions, and objects are included.*

Domain descriptions $\mathsf{D_{EC}}$ are subject to a consistency condition: for every pair consisting of one positive effect axiom $\phi_1 \supset \mathit{Initiates}(a, f, s)$ and one negative effect axiom $\phi_2 \supset \mathit{Terminates}(a, f, s)$ it holds that $\mathsf{D_{EC}} \models \neg(\phi_1 \wedge \phi_2)$. The same holds for every pair consisting of one effect axiom and one release axiom.

Finally, with the help of the above definitions we can specify what it means to solve a planning problem in the Event Calculus:

Definition 2.9 (Planning in the Event Calculus)**.** *Assum given an action domain description* $\mathsf{D_{EC}}$*, an initial situation* ϕ_0*, and a planning goal* γ*. Then planning consists of the abductive reasoning task of finding a narrative* δ *such that the narrative together with the domain description, and the initial situation entails the goal:*

$$\mathit{CIRC}(\bigwedge \mathsf{D_{EC}}; \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}) \wedge \mathit{CIRC}(\phi_0 \wedge \delta; \mathit{Happens}) \wedge \mathsf{EC} \models \gamma.$$

Representing Event Calculus in the UAC We now show how Event Calculus planning domains can be represented in the unifying action calculus. To this end we first recall how to compute circumscription [Lifschitz, 1994].

Definition 2.10 (Computing Circumscription)**.** *Let* ψ *be a formula of the form* $P(\vec{x}) \subset \phi(\vec{x})$*, such that the predicate symbol* P *does not occur in* ϕ*. Then, for the basic circumscription of* P *wrt.* ψ *(*$\mathit{CIRC}(\psi; P)$*), we have that*

$$\mathit{CIRC}(\psi; P) \stackrel{def}{=} (\forall) P(\vec{x}) \equiv \phi(\vec{x}).$$

Circumscribing predicates $P_1, \ldots, P_n$ *in parallel is defined as follows: Let every* P_i *be positive in* ψ*, i.e.* ψ *is equivalent to a formula* ψ' *using only the connectives* $\wedge, \vee, \neg$*, such that each* P_i *occurs in* ψ' *in the scope of an even number of negation signs. Then we have that*

$$\mathit{CIRC}(\psi; P_1, \ldots, P_n) \stackrel{def}{=} \bigwedge_i \mathit{CIRC}(\psi; P_i).$$

Without loss of generality we stipulate that the effect and release axioms in Event Calculus planning domains are represented in their equivalent first order form obtained using definition 2.10. We stipulate the same for any inferred narratives, whether they constitute partial or complete plans. We are now in a position to define a mapping from Event Calculus planning domains to action domains in the unifying action calculus.

Definition 2.11 (Translating Event Calculus Domains into the UAC)**.** *Assume given an Event Calculus domain description* $\mathsf{D_{EC}}$*, and an initial situation* ϕ_0*. These are mapped to the unifying action calculus as follows:*

- *We introduce a UAC signature containing corresponding symbols for the events, fluents, and objects of the Event Calculus domain.*
- *The initial situation* ϕ_0 *is mapped to a initial state axiom* $\phi(0)$*, by replacing every* $\mathit{Init}_P(f)$ *by* $\mathit{Holds}(f, 0)$ *and every* $\mathit{Init}_N(f)$ *by* $\neg \mathit{Holds}(f, 0)$*.*

- *The state constraint (if any) is taken as it is as a domain constraint.*
- *The planning goal γ is also taken as it is.*
- *Foundational axioms $\mathsf{D}_{\mathsf{aux}}$ contain a first order axiomatization of the natural numbers.*
- *For every event $A(\vec{x})$ in the domain we introduce a precondition axiom*

$$(\forall) Poss(A(\vec{x}), s_1, s_2) \equiv Holds(Occurs(A(\vec{x})), s_1) \wedge s_2 = s_1 + 1.$$

- *We include a domain constraint*

$$(\forall)(Holds(Occurs(a_1), s) \wedge Holds(Occurs(a_2), s)) \supset a_1 = a_2,$$

  stipulating that only a single event can occur at any one time.

- *For some ground event a, let $\mathsf{D}^{+}_{\mathsf{EC}}$ be the set of all the respective positive effect axioms, $\mathsf{D}^{-}_{\mathsf{EC}}$ the set of negative effect axioms, and $\mathsf{D}^{?}_{\mathsf{EC}}$ the set of release axioms. Further set $\mathsf{D}^{Eff}_{\mathsf{EC}} = \mathsf{D}^{+}_{\mathsf{EC}} \cup \mathsf{D}^{-}_{\mathsf{EC}} \cup \mathsf{D}^{?}_{\mathsf{EC}}$. Let S be some subset of $\mathsf{D}^{Eff}_{\mathsf{EC}}$. Set*
  - $\mathsf{S}^{+} = \{\ g \mid (\phi \equiv Initiates(a, g, s)) \in \mathsf{S}\}$*;*
  - $\mathsf{S}^{-} = \{\ g \mid (\phi \equiv Terminates(a, g, s)) \in \mathsf{S}\}$*; and*
  - $\mathsf{S}^{?} = \{\ g \mid (\phi \equiv Releases(a, g, s)) \in \mathsf{S}\}$.

  These are the fluents that are potentially initiated, terminated, and released, respectively, by the subset S of effect and release axioms. Further set
  - $\mathsf{Cond}_{\mathsf{S}} = \{\ \phi(s) \mid (\phi(s) \equiv \phi) \in \mathsf{S}\}$*; and*
  - $\mathsf{Cond}_{\bar{\mathsf{S}}} = \{\ \phi(s) \mid (\phi(s) \equiv \phi) \in \mathsf{D}^{Eff}_{\mathsf{EC}} \setminus \mathsf{S}\}$.

  The first of these sets contains those conditions for effects from $\mathsf{D}^{Eff}_{\mathsf{EC}}$ that are contained in S; the second contains those conditions that are not contained in S. Finally, define the following effect axiom in the unifying action calculus:

$$\begin{aligned} &Poss(a, s_1, s_2) \supset \\ &\quad \bigvee_{\mathsf{S} \subseteq \mathsf{D}^{Eff}_{\mathsf{EC}}} \bigwedge_{\mathsf{Cond}_{\mathsf{S}}} \phi(s_1) \wedge \bigwedge_{\mathsf{Cond}_{\bar{\mathsf{S}}}} \neg\phi(s_1) \wedge \\ &\quad (\forall f)[(\bigwedge_{g_i \in \mathsf{S}^{?}} f \neq g_i) \supset \\ &\qquad ((\bigvee_{g_j \in \mathsf{S}^{+}} f = g_j \vee Holds(f, s_1)) \wedge \neg \bigvee_{g_k \in \mathsf{S}^{-}} f = g_k) \equiv Holds(f, s_2)] \end{aligned}$$

The effect axioms obtained in this manner for each ground event $A(\vec{t})$ in the Event Calculus planning domain can then be combined in a single UAC effect axiom for the parametric action $A(\vec{x})$.

Please note that the effect axioms defined above do not adhere strictly to the definition of an effect axiom in the UAC (cf. definition 2.2). This deviation is necessary, however, if we want to express that some fluents are exempt from the frame assumption, i.e. there are occluded fluents. Using the same trick it is possible to exclude fluents from the frame assumption also in Fluent and Situation Calculus.

We have defined a mapping from Event Calculus planning domains to domain axiomatizations D in the UAC. It remains to show that these two formulations admit the same solutions. To this end define a totally ordered narrative in the UAC exactly as in the Event Calculus — cf. definition 2.7 — only using $\text{Holds}(\text{Occurs}(a), s)$ atoms, instead of $\text{Happens}(a, s)$ atoms.

Proposition 2.1 (Correctness of the Translation)**.** *Let an Event Calculus planning domain D_{EC} with goal γ be mapped to a UAC domain axiomatization D. There exists a narrative δ_{EC} solving the Event Calculus planning problem if and only if there exists a narrative δ_{UAC} such that $\mathsf{D} \cup \delta_{UAC} \models \gamma$.*

Please note that the version of the Event Calculus that we have defined does not provide for event preconditions, exactly like the Event Calculus for planning presented in [Shanahan, 2000]. This omission is due to the fact that for not completely specified initial situations the minimization of Happens may lead to unintuitive results [Miller and Shanahan, 2002]. Please also note that extending our version of the Event Calculus to include event preconditions wrt. completely specified initial situations poses no technical difficulties.

2.3.4 Modularity of Domain Axiomatizations

In this section we recall the notion of modularity of domain axiomatizations. This notion underlies our own implementation work discussed in chapter 4 and 5 below.

The problem of modularity of domain axiomatizations arises from the fact that axiomatizations of action domains combine different categories of formulas which serve different purposes [Herzig and Varzinczak, 2007]. *Domain constraints* describe static properties which hold in all states; *precondition axioms* define the conditions for actions to be applicable; and *effect axioms* define the consequences of actions. As a uniform logical theory, however, a domain axiomatization may easily give rise to dependencies among the different kinds of axioms: Effect formulas can entail implicit preconditions, domain constraints can entail implicit effects, etc. Implementations like Golog or Flux, on the other hand, rely on the assumption that dependencies like these do not exist, i.e. that the domain axiomatization is modular. The reason

is that, for the sake of efficiency, the implementations use domain axiomatizations in a modular fashion. Agents use the domain constraints only when they initialize their world model, they check the applicability of an action merely against the precondition axioms, and they update their world model entirely on the basis of the effect axioms. Agent programs would be much less efficient if the entire domain theory had to be taken into account for each specific reasoning task.

In [Thielscher, 2007] conditions for modularity have been presented against which a domain axiomatization can be checked. As the main result, there it has been proven that the class of sequential and ramification-free domain axiomatizations (cf. definition 3.8) are guaranteed to be free of dependencies if they satisfy these conditions. Our own implementations are based on fragments of the Fluent Calculus in the UAC only. Hence here we recall only the material relevant for modularity in the Fluent Calculus. Modularity in the Fluent Calculus is easier to check because the domain axiomatizations are always sequential and ramification-free.

We next recall the definition of implicit domain constraints, preconditions, and effects. To this end, we introduce the following notation for a given action $A(\vec{x})$: In a domain axiomatization with precondition axioms $\mathsf{D}_{\mathsf{Poss}}$, by $\mathsf{D}_{\mathsf{Poss}\,A}$ we denote the one precondition axiom which is for $A(\vec{x})$, with $\pi_A[s]$ being its right hand side as usual. Likewise, if $\mathsf{D}_{\mathsf{Effects}}$ are the effect axioms, then by $\mathsf{D}_{\mathsf{Effects}\,A}$ we denote the one for action $A(\vec{x})$.

Definition 2.12. *Consider a domain axiomatization Σ consisting of domain constraints D_{dc}, precondition axioms $\mathsf{D}_{\mathsf{Poss}}$, and effect axioms $\mathsf{D}_{\mathsf{Effects}}$.*

1. *The domain axiomatization is* free of implicit domain constraints *if for every state formula $\delta[t]$,*
$$\Sigma \models \delta[t]$$
implies $\mathsf{D}_{\mathsf{dc}} \models \delta[t]$.

2. *The domain axiomatization is* free of implicit preconditions *if for every action $A(\vec{x})$ and state formula $\pi[s]$,*
$$\Sigma \models \mathit{Poss}(A(\vec{x}), s, t) \supset \pi[s]$$
implies $\mathsf{D}_{\mathsf{dc}} \cup \mathsf{D}_{\mathsf{Poss}\,A} \models \mathit{Poss}(A(\vec{x}), s, t) \supset \pi[s]$.

3. *The domain axiomatization is* free of implicit effects *if for every action $A(\vec{x})$ and state formula $\varepsilon[t]$, if*
$$\Sigma \models \mathit{Poss}(A(\vec{x}), s, t) \supset \varepsilon[t]$$
then $\mathsf{D}_{\mathsf{dc}}[S] \cup \mathsf{D}_{\mathsf{Poss}\,A}[S] \cup \mathsf{D}_{\mathsf{Effects}\,A}[S, T] \models \mathit{Poss}(A(\vec{x}), S, T) \supset \varepsilon[T]$, where S, T are constants of sort Time.

Put in words, an implicit domain constraint is a domain constraint which is entailed by the entire domain axiomatization but which cannot be derived from the given domain constraints D_{dc} alone. An implicit precondition is entailed by the entire domain axiomatization but does not follow from the precondition axioms alone in a state that satisfies the domain constraints. The rationale behind this definition is the following: Given a state that satisfies the domain constraints D_{dc}, the precondition axiom for an action A alone should suffice to entail all executability conditions for this action. Finally, an implicit effect follows from the entire domain axiomatization but not from an effect axiom alone in a state that satisfies the preconditions of an action and the domain constraints. The rationale behind this definition is this: Given a state that satisfies both the domain constraints D_{dc} and the preconditions of an action A, the instantiated effect axiom for this action alone should suffice to infer everything that can be concluded of the resulting state.

The following two conditions are enough to guarantee that a Fluent Calculus domain axiomatization is free of implicit dependencies. Informally speaking, the first condition, (4.1), essentially says that for every state at some time S which is consistent with the domain constraints and in which an action $A(\vec{x})$ is applicable, the condition $\Phi_i[S]$ for at least one case i in the effect axiom for A holds. Condition (4.2) requires that any possible update leads to a state that satisfies the domain constraints.

Definition 2.13. *Let S, T be constants of sort* Time. *A domain axiomatization with domain constraints D_{dc}, precondition axioms $\mathsf{D}_{\mathsf{Poss}}$, and effect axioms $\mathsf{D}_{\mathsf{Effects}}$ is called* modular *if the following holds for every action $A(\vec{x})$ with effect axiom (2.2): There exists $i = 1, \ldots, k$ such that*

$$\models \mathsf{D}_{\mathsf{dc}}[S] \wedge \pi_A[S] \wedge (\exists \vec{y}_i)\Phi_i[S], \tag{2.1}$$

and for every such i,

$$\models \mathsf{D}_{\mathsf{dc}}[S] \wedge \pi_A[S] \wedge \eta_i[S, T] \supset \mathsf{D}_{\mathsf{dc}}[T] \tag{2.2}$$

We next recall one of the main results from [Thielscher, 2007], which says that modular domain axiomatizations are free of implicit domain constraints, preconditions, and effects.

Theorem 2.6. *If a sequential and ramification-free domain axiomatization is modular then it is also free of implicit domain constraints, preconditions, and effects.*

2.3.5 Reasoning with Action Theories

In the chapters 4 and 5 we will discuss issues related to the practical implementation of the theoretical ALP framework. Let us already here draw a distinction between

two fundamentally different ways of implementing reasoning with action theories: An implementation may be based on progression, or it may be based on regression.

The *progression approach* is very similar to that of traditional relational database systems. We view the initial state formula $\mathsf{D}_{\mathsf{Init}}$ as some kind of database. Next we view the effects of an action as a database update — we compute an updated database. Finally, queried state properties are evaluated against the current database just like any database query. The action programming language Flux is a prominent example of an implementation based on the idea of progression.

In the *regression approach*, on the other hand, we never update the initial database. Instead it only remembers the sequence of executed actions. For answering a query that is posed after executing a sequence of actions the following approach is taken: The query is iteratively rewritten by taking the effects of the previously executed actions into account. Finally, the completely rewritten query is evaluated against the initial database. The most prominent example of an action programming language based on regression is Golog.

Both implementations described in this thesis are based on the progression approach.

2.4 Description Logics

Description Logics (DLs) are a recently very successful branch of logic-based knowledge representation formalisms. They (usually) go considerably beyond the expressivity of propositional logic, while at the same time they avoid the undecidability of full first order logic. Although the complexity of the reasoning problems in DLs can appear to be daunting the available highly optimized reasoners[6] have empirically proven to be sufficiently efficient for applications.

In this section we introduce those aspects of Description Logics that underpin our implementation of a fragment of the ALP framework discussed in chapter 5. In a first subsection we introduce the basics of static DL-based knowledge representation. In a second subsection we briefly recall the theoretical insights on dynamic DL-based knowledge representation from [Liu et al., 2006] that underlie our implementation work. The definitive textbook on Description Logics is [Baader et al., 2003].

The Description Logics considered in this thesis do not incorporate expressive features that go beyond first order logic: Thus they can all be considered as fragments of first order logic. However, the DL community has established independent notational conventions that we will adhere to, even if they are at odds with our own notational conventions laid out in section 2.1. The reader interested in the minutiae of the relation between Description Logics and first order logic is referred to [Borgida, 1996].

[6] A list of DL reasoners is available at `http://www.cs.man.ac.uk/~sattler/reasoners.html`.

2.4.1 Basic Description Logics

In DLs, knowledge is represented with the help of concepts (unary predicates) and roles (binary predicates). Complex concepts and roles are inductively defined starting with a set $\mathsf{N_C}$ of *concept names*, a set $\mathsf{N_R}$ of *role names*, and a set $\mathsf{N_I}$ of *individual names*. The expressiveness of a DL is determined by the set of available *constructors* to build *concepts* and *roles*. The concept and role constructors of the DLs $\mathcal{ALCO}^{@}$, $\mathcal{ALCIO}^{@}$, $\mathcal{ALCO}^{+}$ and $\mathcal{ALCO}^{+}$ that form the base of our work on the implementation of ABox update discussed in chapter 5 are shown in Table 2.1, where C, D are concepts, q, r are roles, and a, b are individual names. The DL that allows only for negation, conjunction, disjunction, and universal and existential restrictions is called $\mathcal{ALC}$. By adding nominals $\mathcal{O}$, we obtain $\mathcal{ALCO}$, which is extended to $\mathcal{ALCO}^{@}$ by the @-constructor from hybrid logic [Areces and de Rijke, 2001], and to $\mathcal{ALCO}^{+}$ by the Boolean constructors on roles and the nominal role [Liu et al., 2006].[7] Adding inverse roles to $\mathcal{ALCO}^{@}$ or $\mathcal{ALCO}^{+}$ we obtain $\mathcal{ALCIO}^{@}$ and $\mathcal{ALCIO}^{+}$. We will use $\top$ ($\bot$) to denote arbitrary tautological (unsatisfiable) concepts and roles. By $\mathsf{sub}(\phi)$ we denote the set of all subconcepts and subroles of a concept or role ϕ, respectively, where, e.g., the subconcepts of a complex concept are all the concept names occuring in it. By $\mathsf{Obj}(\mathcal{A})$ we denote all the individuals that occur in $\mathcal{A}$.

Name	Syntax	Semantics
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
universal restriction	$\forall r C$	$\{x \mid \forall y.((x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}})\}$
existential restriction	$\exists r C$	$\{x \mid \exists y.((x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}})\}$
nominal	$\{a\}$	$\{a^{\mathcal{I}}\}$
@ constructor	$@_a C$	$\Delta^{\mathcal{I}}$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, and $\emptyset$ otherwise
inverse role	r^-	$(r^{\mathcal{I}})^{-1}$
role negation	$\neg r$	$(\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) \setminus r^{\mathcal{I}}$
role conjunction	$q \sqcap r$	$q^{\mathcal{I}} \cap r^{\mathcal{I}}$
role disjunction	$q \sqcup r$	$q^{\mathcal{I}} \cup r^{\mathcal{I}}$
nominal role	$\{(a,b)\}$	$\{(a^{\mathcal{I}}, b^{\mathcal{I}})\}$

Table 2.1: Syntax and semantics of $\mathcal{ALCO}^{@}$ and $\mathcal{ALCO}^{+}$.

The semantics of concepts and roles is given via *interpretations* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. The

[7] Our version of $\mathcal{ALCO}^{+}$ is equivalent to the one found in [Liu et al., 2006], but admits a more streamlined presentation.

domain $\Delta^{\mathcal{I}}$ is a non-empty set and the *interpretation function* $\cdot^{\mathcal{I}}$ maps each concept name $A \in \mathsf{N_C}$ to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$, each role name $r \in \mathsf{N_R}$ to a binary relation $r^{\mathcal{I}}$ on $\Delta^{\mathcal{I}}$, and each individual name $a \in \mathsf{N_I}$ to an individual $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The interpretation function $\cdot^{\mathcal{I}}$ is inductively extended to complex concepts and roles as shown in Table 2.1.

An *ABox assertion* is of the form $C(a)$, $r(a,b)$, or $\neg r(a,b)$ with r a role, C a concept and a, b individual names. A *classical ABox*, or an *ABox* for short, is a finite conjunction of ABox assertions. A *Boolean ABox* is a Boolean combination of ABox assertions. For convenience we will also sometimes represent classical and Boolean ABoxes as finite sets of assertions by breaking the top-level conjunctions. An interpretation $\mathcal{I}$ is a *model* of an assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$. $\mathcal{I}$ is a model of an assertion $r(a,b)$ (resp. $\neg r(a,b)$) if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ (resp. $(a^{\mathcal{I}}, b^{\mathcal{I}}) \notin r^{\mathcal{I}}$). A model of a (Boolean) ABox is defined in the obvious way. We use $M(\mathcal{A})$ to denote the set of models of a Boolean ABox $\mathcal{A}$. A (Boolean) ABox $\mathcal{A}$ is *consistent* if $M(\mathcal{A}) \neq \emptyset$. Two (Boolean) ABoxes $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent*, denoted by $\mathcal{A} \equiv \mathcal{A}'$, if $M(\mathcal{A}) = M(\mathcal{A}')$. An assertion α is *entailed* by a Boolean ABox $\mathcal{A}$, written as $\mathcal{A} \models \alpha$, if $M(\mathcal{A}) \subseteq M(\{\alpha\})$. Classical $\mathcal{ALCO}^{@}$ ABoxes can equivalently be compiled to Boolean $\mathcal{ALCO}$ ABoxes (and vice versa) — the translation in the first direction is exponential, in the other direction it is linear [Liu et al., 2006]. *Consistency checking* and *entailment* for classical ABoxes are standard inference problems and supported by all DL reasoners, while, to the best of our knowledge, no state of the art reasoner directly supports these inferences for Boolean ABoxes. Reasoning in $\mathcal{ALCO}^{+}$ and $\mathcal{ALCIO}^{+}$ is NExpTime complete [Tobies, 2001]; for $\mathcal{ALCO}^{@}$ it is PSpace complete [Areces et al., 1999], and for $\mathcal{ALCIO}^{@}$ we only know that already reasoning with $\mathcal{ALCIO}$ ABoxes is ExpTime complete [Schaerf, 1994].

Another very useful feature of DLs are TBoxes that allow us to introduce abbreviations for complex concepts. A TBox $\mathcal{T}$ is a finite set of concept definitions of the form $A \equiv C$, where A is a concept name (called a defined concept) and C is a complex concept. A TBox is acyclic if it does not contain multiple or cyclic definitions.

A DL Knowledge Base is a pair $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, with TBox $\mathcal{T}$ and ABox $\mathcal{A}$; the semantics of Knowledge Bases is defined in the obvious way. Reasoning wrt. an acyclic TBox in a knowledge base can always be reduced to reasoning wrt. the empty TBox by unfolding the definitions [Baader et al., 2003].

The following, more general notion of a TBox is also sometimes used: A TBox is a set of concept inclusions, where a concept inclusion is an expression $C \sqsubseteq D$ with first order reading $(\forall x)C(x) \supset D(x)$. A concept definition $C \equiv D$ can be seen as two concept inclusions, $C \sqsubseteq D$ and $D \sqsubseteq C$. In this thesis we use the previous, less general definition of a TBox; at one point we will discuss issues related to concept inclusions, though.

Later on, it will be useful to have a notion of the size of an ABox at our disposal.

We choose the following definition:

Definition 2.14 (ABox Size)**.** *The size* $|\mathcal{A}|$ *of an ABox is* $\Sigma_{\alpha\in\mathcal{A}}|\alpha|$*, the sum of the sizes of the respective assertions. For Boolean assertions* $\alpha(\vee,\wedge)\beta$ *we define the size as* $1+|\alpha|+|\beta|$*. The size* $|C(a)|$ *of a concept assertion is* $1+|C|$*, and the size* $|r(a,b)|$ *of a role assertion is* $2+|r|$*. The size of a concept is*

- 1*, for concept names and nominals;*
- $1+|C|+|D|$ *for* $C\sqcap D, C\sqcup D$*;*
- $1+|r|+|C|$ *for* $\exists r.C, \forall r.C$*; and*
- $1+|C|$ *for* $\neg C$*.*

The size of a roles is defined analogously, with the convention that a nominal role is of size 2*. Finally, the size of TBox elements* $C\sqsubseteq D$ *or* $C\equiv D$ *is defined as* $1+|C|+|D|$*.*

2.4.2 ABox Update

An ABox can be used to represent knowledge about the state of some world. An *update* contains information on changes that have taken place in that world. In [Liu et al., 2006], a method for updating DL ABoxes has been developed, and in [Drescher and Thielscher, 2007] we have shown that this notion of an update conforms with the semantics employed by Fluent and Situation Calculus.

This thesis is accompanied by a sister thesis [Liu, 2009] that studies ABox updates in depth. Here we only recall those aspects of ABox update in detail that underlie our own implementation. However, we evaluate our implementation techniques for the updates introduced below by comparing them to an implementation of the more sophisticated, so-called projective updates of [Liu, 2009] in section 5.2.2 below. Hence we conclude this section by an cursory overview of the projective ABox updates introduced in [Liu, 2009].

Updates

Intuitively, an update $\mathcal{U}$ describes the changes that have happened in the world. The ABox updates considered in [Liu et al., 2006] are conditional, and they work on the DLs between $\mathcal{ALCO}^{@}$ and $\mathcal{ALCQIO}^{@}$, or $\mathcal{ALCO}^{+}$ and $\mathcal{ALCQIO}^{+}$, respectively. Since our prototypical implementation supports only unconditional updates in the DLs $\mathcal{ALCO}^{@}$ and $\mathcal{ALCIO}^{@}$ (or $\mathcal{ALCO}^{+}$ and $\mathcal{ALCIO}^{+}$, respectively) we start by recalling only the respective material. Our implementation does not support counting quantifiers (in the guise of the qualified number restrictions $\mathcal{Q}$) because they lead to updated ABoxes that are hard to manage in practice.

The following definition of an unconditional update as a consistent ABox $\mathcal{U}$ says for every literal $\delta(\vec{t}) \in \mathcal{U}$ that this literal holds after the change of the world state:

Definition 2.15 (Update)**.** *An* update $\mathcal{U}$ *is a finite consistent set of DL literals, i.e. each* $\delta(\vec{t}) \in \mathcal{U}$ *is of the form* $A(a)$, $\neg A(a)$, $r(a,b)$, *or* $\neg r(a,b)$ *with* A *a concept name,* r *a role name, and* a, b *individual names.*

The formal semantics of updates given in [Liu et al., 2006] defines, for every interpretation $\mathcal{I}$, a successor interpretation $\mathcal{I}^{\mathcal{U}}$ obtained by minimally changing this model according to the update. In particular, there is no difference in the interpretation of individual names between the original and the updated interpretation. Given an ABox $\mathcal{A}$, all its models are considered to be possible current states of the world. The goals is then to find an updated ABox $\mathcal{A}*\mathcal{U}$ that has exactly the successor of the models of $\mathcal{A}$ as its models, i.e., $\mathcal{A}*\mathcal{U}$ must be such that $M(\mathcal{A}*\mathcal{U}) = \{\mathcal{I}^{\mathcal{U}} \mid \mathcal{I} \in M(\mathcal{A})\}$. In general, such an updated ABox need not exists. This semantics of ABox updates is based on the possible models approach of Winslett [Winslett, 1988]. It should be noted here that for the deterministic updates under consideration Winslett semantics is uncontroversial, even though it does not extend to updates with non-deterministic or indirect effects (cf., e.g., [Doherty et al., 1998]) — these latter effects are also called ramifications.

Admitting the inverse role constructor in updates does not add expressivity, but complicates the presentation. Hence, we refrain from doing so. But, for presenting the construction of updated ABoxes, it will be helpful to assume a certain normal form on ABoxes: Without loss of generality we assume that the inverse role constructor does not occur in role assertions. Likewise we assume that it does not occur in nested form in the quantifier restrictions $\exists r.C$ and $\forall r.C$ — e.g., we assume that $\exists r^{--}.C$ is replaced by $\exists r.C$.

Updated ABoxes

We continue by recapitulating the construction of updated ABoxes.

The minimal DLs that contain both the basic DL $\mathcal{ALC}$ and are closed under ABox updates are $\mathcal{ALCO}^{@}$ and Boolean $\mathcal{ALCO}$. For $\mathcal{ALCO}^{@}$ and $\mathcal{ALCIO}^{@}$, updated ABoxes are exponential in the size of the original ABox and the update. The DLs $\mathcal{ALCO}^{+}$ and $\mathcal{ALCIO}^{+}$ admit updated ABoxes that are exponential in the size of the update, but polynomial in the size of the original ABox.

We continue by introducing additional notation: A simple ABox $\mathcal{D}$ is called a *diagram for* $\mathcal{U}$ if it is a maximal consistent subset of $L_{\mathcal{U}}$, where $L_{\mathcal{U}} = \{\psi, \neg\psi \mid \psi \in \mathcal{U}\}$ is the set of *literals* over $\mathcal{U}$. Intuitively, a diagram gives a complete description of the part of a model of $\mathcal{A}$ that is "relevant" for the update $\mathcal{U}$. Let $\mathfrak{D}$ be the set of all diagrams for $\mathcal{U}$ and consider for $\mathcal{D} \in \mathfrak{D}$ the set $\mathcal{D}_{\mathcal{U}} := \{\psi \mid \neg\psi \in \mathcal{D} \text{ and } \psi \in \mathcal{U}\}$. Thus, $\mathcal{D}_{\mathcal{U}}$ contains exactly those assertions from $\mathcal{U}$ that do not hold in $\mathcal{D}$. Now the

$$
\begin{array}{lll}
A^{\mathcal{U}} = & (A \sqcup \bigsqcup_{\neg A(a) \in \mathcal{U}} \{a\}) \sqcap & \\
& \neg(\bigsqcup_{A(a) \in \mathcal{U}} \{a\}) & \\
& & \\
r^{\mathcal{U}} = & (r \sqcup \bigsqcup_{\neg r(a,b) \in \mathcal{U}} \{(a,b)\}) \sqcap & \\
& \neg(\bigsqcup_{r(a,b) \in \mathcal{U}} \{(a,b)\}) & \\
& & \\
(r^-)^{\mathcal{U}} = & (r^{\mathcal{U}})^- & \\
\{a\}^{\mathcal{U}} = & \{a\} & \{(a,b)\}^{\mathcal{U}} = \quad \{(a,b)\} \\
(\neg C)^{\mathcal{U}} = & \neg C^{\mathcal{U}} & (\neg r)^{\mathcal{U}} = \quad \neg r^{\mathcal{U}} \\
(C \sqcap D)^{\mathcal{U}} = & C^{\mathcal{U}} \sqcap D^{\mathcal{U}} & (r \sqcap q)^{\mathcal{U}} = \quad r^{\mathcal{U}} \sqcap q^{\mathcal{U}} \\
(C \sqcup D)^{\mathcal{U}} = & C^{\mathcal{U}} \sqcup D^{\mathcal{U}} & (r \sqcup q)^{\mathcal{U}} = \quad r^{\mathcal{U}} \sqcup q^{\mathcal{U}} \\
(\exists r.C)^{\mathcal{U}} = & \exists r^{\mathcal{U}}.C^{\mathcal{U}} & (\forall r.C)^{\mathcal{U}} = \quad \forall r^{\mathcal{U}}.C^{\mathcal{U}}
\end{array}
$$

Figure 2.1: Constructing $C^{\mathcal{U}}$ and $r^{\mathcal{U}}$ for $\mathcal{ALCO}^+$ and $\mathcal{ALCIO}^+$

$$
\begin{array}{ll}
(@_i C)^{\mathcal{U}} = & @_i C^{\mathcal{U}} \\
(\exists r.C)^{\mathcal{U}} = & (\bigsqcap_{a \in \mathsf{Obj}(\mathcal{U})} \neg\{a\} \sqcap \exists r.C^{\mathcal{U}}) \sqcup \\
& \exists r.(\bigsqcap_{a \in \mathsf{Obj}(\mathcal{U})} \neg\{a\} \sqcap C^{\mathcal{U}}) \sqcup \\
& \bigsqcup_{a,b \in \mathsf{Obj}(\mathcal{U}), r(a,b) \notin \mathcal{U}} (\{a\} \sqcap \exists r.(\{b\} \sqcap C^{\mathcal{U}})) \sqcup \\
& \bigsqcup_{\neg r(a,b) \in \mathcal{U}} (\{a\} \sqcap @_b C^{\mathcal{U}}) \\
& \\
(\forall r.C)^{\mathcal{U}} = & (\bigsqcup_{a \in \mathsf{Obj}(\mathcal{U})} \{a\} \sqcup \forall r.C^{\mathcal{U}}) \sqcap \\
& \forall r.(\bigsqcup_{a \in \mathsf{Obj}(\mathcal{U})} \{a\} \sqcup C^{\mathcal{U}}) \sqcap \\
& \bigsqcap_{a,b \in \mathsf{Obj}(\mathcal{U}), r(a,b) \notin \mathcal{U}} (\neg\{a\} \sqcup \forall r.(\neg\{b\} \sqcup C^{\mathcal{U}})) \sqcap \\
& \bigsqcap_{\neg r(a,b) \in \mathcal{U}} (\neg\{a\} \sqcup @_b C^{\mathcal{U}})
\end{array}
$$

Figure 2.2: Constructing $C^{\mathcal{U}}$ for $\mathcal{ALCO}^{@}$ and $\mathcal{ALCIO}^{@}$

updated ABox $\mathcal{A}'$ is defined as the following disjunction, containing one disjunct for every diagram $\mathcal{D}$ for $\mathcal{U}$:

Proposition 2.2 (Updated ABox for $\mathcal{ALCO}^+$ and $\mathcal{ALCIO}^+$)**.** *Let the updated concept (role) $\alpha^{\mathcal{X}}$ be obtained by the construction defined in Figure 2.1. Let the ABox $\mathcal{A}'$ be defined as*

$$\mathcal{A}' \;=\; \bigvee_{\mathcal{D} \in \mathfrak{D}} \bigwedge \mathcal{A}^{\mathcal{D}_{\mathcal{U}}} \cup \mathcal{D}_{\mathcal{U}} \cup (\mathcal{D} \setminus \mathcal{D}_{\mathcal{U}}^{\neg}), \tag{2.3}$$

where the ABox $\mathcal{A}^{\mathcal{X}}$ is defined as $\mathcal{A}^{\mathcal{X}} = \{\alpha^{\mathcal{X}}(\vec{t}) \mid \alpha(\vec{t}) \in \mathcal{A}\}$, and the set $\mathcal{D}_{\mathcal{U}}^{\neg}$ as $\mathcal{D}_{\mathcal{U}}^{\neg} = \{\neg\varphi \mid \varphi \in \mathcal{D}_{\mathcal{U}}\}$. Then $\mathcal{A} * \mathcal{U} \equiv \mathcal{A}'$.

The DLs $\mathcal{ALCO}^{@}$ and $\mathcal{ALCIO}^{@}$ lack role operators, and, hence, the construction of the updated quantifier concepts is complicated — it is depicted in Figure 2.2, as is the construction for the @-constructor. For the Boolean concept constructors and nominals the construction is as in $\mathcal{ALCO}^+$. In Figure 2.2, we omit the construction

of existential and universal restrictions that use the inverse role constructor. Let us emphasize that in Figure 2.2 the symbol r denotes a role name, and hence not an inverse role. Then recall that we assume that the inverse role constructor only occurs non-nestedly, and in direct combination with a role name: The respective constructions for quantifier restrictions with inverse roles can be obtained from those depicted by appropriately swapping individual names.

Proposition 2.3 (Updated ABox for $\mathcal{ALCO}^{@}$ and $\mathcal{ALCIO}^{@}$)**.** *In this setting the ABox $\mathcal{A}^{\mathcal{X}}$ is defined as*

$$\begin{aligned}\mathcal{A}^{\mathcal{X}} = \quad & \{C^{\mathcal{X}}(a) \mid C(a) \in \mathcal{A}\} \cup \{r(a,b) \mid r(a,b) \in \mathcal{A} \wedge \neg r(a,b) \notin \mathcal{X}\} \cup \\ & \{\neg r(a,b) \mid \neg r(a,b) \in \mathcal{A} \wedge r(a,b) \notin \mathcal{X}\}.\end{aligned}$$

Let $\mathcal{A}'$ be as defined in (2.3). Then $\mathcal{A} * \mathcal{U} \equiv \mathcal{A}'$.

To see how the construction of updated ABoxes works consider the following example:

Example 2.2 (Updated ABox). Let the ABox $\mathcal{A} = \{A(a)\}$ be updated with $\mathcal{U} = \{\neg A(a)\}$. Following Propositions 2.2 and 2.3 we obtain the (highly redundant) updated ABox

$$\{(A(a) \wedge \neg A(a)) \vee ((A \sqcup \{a\})(a) \wedge \neg A(a))\},$$

which can be simplified to $\{\neg A(a)\}$. Intuitively, there is one disjunct $(\mathcal{A} \cup \mathcal{U})$ for the case that the update already held before the update, and one disjunct $(\mathcal{A}^{\mathcal{U}} \cup \mathcal{U})$ for the case that its negation did.

Let us provide some intuition for why the updated ABoxes in $\mathcal{ALCO}^{@}$ (and its extensions) are exponential in the update and the original ABox, whereas in $\mathcal{ALCO}^{+}$ they are exponential in the update only: In $\mathcal{ALCO}^{+}$, due to the presence of Boolean and nominal concept and role constructors, we can use exactly the same update techniques as for role literals for role and concept literals in the update. In $\mathcal{ALCO}^{@}$, because of the absence of role constructors, for role literals in the update we have to use a different update construction than for concept literals. In fact, if updates in $\mathcal{ALCO}^{@}$ contain no role literals then the updated ABoxes are exponential in the update only [Liu et al., 2006]. The exponential blowup caused by role assertions in the updated stems from the duplication of the concept $C^{\mathcal{U}}$ when updating the quantifier restrictions $\exists r.C$ and $\forall r.C$ as witnessed by the construction in figure 2.2. An important result from [Liu et al., 2006] is that in the case of ABoxes iteratively updated with a sequence of updates the exponential blowups do not add up.

Conditional Updates

In [Liu et al., 2006] it has also been shown how the construction of updated ABoxes can be extended to conditional updates. Here we recall the respective definitions

that we need to prove our result on a UAC semantics for ABox update in section 5.1 below.

A conditional updated is defined as a finite set of pairs ϕ/ψ to be read as follows: For every model $\mathcal{I}$ of the original ABox, the effect specified by the literal ψ materializes in the updated interpretation $\mathcal{I}'$ if and only if the assertion ϕ is satisfied by $\mathcal{I}$. Formally, this semantics of update is defined as follows:

Definition 2.16 (Conditional Interpretation Update)**.** *Let $\mathcal{U}$ be a conditional update and $\mathcal{I}$, $\mathcal{I}'$ interpretations such that $|\Im|^{\mathcal{I}} = |\Im|^{\mathcal{I}'}$ and $\mathcal{I}$ and $\mathcal{I}'$ agree on the interpretation of individual names. Then $\mathcal{I}'$ is the result of updating $\mathcal{I}$ with $\mathcal{U}$, written $\mathcal{I} \Longrightarrow_{\mathcal{U}} \mathcal{I}'$, if the following holds for all concept names $C \in \mathsf{N_C}$ and role names $R \in \mathsf{N_R}$:*

$$\begin{aligned} C^{\mathcal{I}'} &= (C^{\mathcal{I}} \cup \{\ I^{\mathcal{I}} \mid \varphi/C(I) \in \mathcal{U} \wedge \mathcal{I} \vDash \varphi\}) \\ &\qquad \setminus \{\ I^{\mathcal{I}} \mid \varphi/\neg C(I) \in \mathcal{U} \wedge \mathcal{I} \vDash \varphi\} \text{and} \\ R^{\mathcal{I}'} &= (R^{\mathcal{I}} \cup \{\ (I_1^{\mathcal{I}}, I_2^{\mathcal{I}}) \mid \varphi/R(I_1, I_2) \in \mathcal{U} \wedge \mathcal{I} \vDash \varphi\}) \\ &\qquad \setminus \{\ (I_1^{\mathcal{I}}, I_2^{\mathcal{I}}) \mid \varphi/\neg R(I_1, I_2) \in \mathcal{U} \wedge \mathcal{I} \vDash \varphi\}. \end{aligned}$$

Let in the following $\mathcal{M}(\mathcal{A})$ denote the set of all models of an ABox $\mathcal{A}$. Then a conditionally updated ABox is defined as follows:

Definition 2.17 (Conditionally Updated ABox)**.** *For an ABox $\mathcal{A}$ and a conditional update $\mathcal{U}$ the updated ABox $\mathcal{A}'$ is defined model-theoretically such that:*

$$\mathcal{M}(\mathcal{A}') = \{\ \mathcal{I}' \mid \mathcal{I} \in \mathcal{M}(\mathcal{A})\ \wedge \mathcal{I} \Longrightarrow_{\mathcal{U}} \mathcal{I}'\}.$$

The formal definitions for unconditional interpretation and ABox updates are obtained as the respective special cases.

In chapter 5 below we will give an action calculus semantics for conditional ABox update based on the above definitions. Our prototypical implementation of ABox update also discussed in chapter 5 below does not cover conditional updates, though. This choice is motivated by the observation that it is already difficult to control the non-determinism present in unconditional ABox updates. Hence, here we refrain from recapitulating the construction of conditionally updated ABoxes from [Liu et al., 2006].

ABox Update and (Non-)Unique Names

Let us conclude this introduction by some remarks on ABox update and the unique name assumption. In principle, ABox update as introduced above works both with and without it. However, the original and the updated interpretation have to agree on the interpretation of individual names (cf. definition 2.16). Now, if the input ABox $\mathcal{A}$ and the update $\mathcal{U}$ entail different (dis-)equalities between individual names

this can lead to problems. For example, the input ABox $\mathcal{A} = \{\{i\}(j)\}$ cannot be updated to a consistent ABox with the update $\mathcal{U} = \{C(i), \neg C(j)\}$.

In order to overcome this issue we propose to parametrize ABox update by a set $\mathcal{EQ}$ of (dis-)equalities between individual names (this set $\mathcal{EQ}$ need not be maximal). Then, in a next step, we stipulate that no ABox or (possibly conditional) update may entail a (dis-)equality between individual names that is not already contained in $\mathcal{EQ}$. Adopting the unique name assumption is a particular simple instance of this approach, resulting in a maximal set of dis-equalities $\mathcal{EQ}$.

Observe that this issue is closely related to the modularity of action theories as introduced in section 2.3.4 above.

ABox Update in the Presence of TBoxes

If the ABox $\mathcal{A}$ is augmented by a acyclic terminology $\mathcal{T}$ we take that to mean that the TBox holds at all time-points, i.e. we interpret the TBox as a domain constraint. If we reconsider the construction of updated ABoxes we see that we have to place the following restrictions on the usage of defined concepts in ABoxes and conditional updates:

- Defined concepts in the ABox are assumed be expanded to primitive concepts — otherwise ABox update would give wrong results.
- Defined concepts must not be used in the effect ψ of a conditional update ϕ/ψ — otherwise effects would not be limited to literals, and ABox update would not work.
- Defined concepts may freely be used in the condition ϕ of a conditional update ϕ/ψ.

Logical, Approximate, and Projective Updates

In the accompanying sister thesis [Liu, 2009] by Hongkai Liu the ABox update problem has been studied in much more detail. Here, we briefly recall some results from that thesis that are relevant for this thesis.

In the terminology of [Liu, 2009] the updates recapitulated above are logical updates. Another type of update studied in [Liu, 2009] are the so-called projective updates. More precisely, [Liu, 2009] introduces three types of update ABoxes:

- Logical updates, that have exactly one updated model for each model of the original ABox.
- Approximate updates, that prove the same things as logical updates, but do not have the same models.

- Projective updates, that extend the signature of the original ABox, but prove the same things as logical updates if projected to the original signature.

Very roughly, the idea underlying projective updates is

- to first introduce temporalized concept and role names — e.g., A_1 is used to denote the state of the concept A after the first update; and
- then to use these temporalized concept and role names in a knowledge base $\mathcal{K}$ consisting of TBox and ABox that together describe the current state of the world.

The most important result wrt. projective updates from [Liu, 2009] is that the knowledge base $\mathcal{K}$ is polynomial in both the input ABox and the update, in striking contrast to the logical updates introduced above. However, their more complicated construction makes them less amenable to optimization techniques in the implementation as we will further discuss in section 5.2.2 below.

3 Action Logic Programs

The purpose of action logic programs (ALPs) is to define heuristics and strategies for solving fully general planning problems on top of action domain axiomatizations D. A distinguishing feature of ALPs is their independence of the specific representation formalism used to axiomatize the underlying action domain. Thus ALPs can be combined with a variety of different action calculi. To demonstrate this, we base ALPs on action domains formulated in the recently proposed unifying action calculus [Thielscher, 2007].

ALPs are definite logic programs augmented with two special predicates, one — written `do` — for executing an action and one — written `?` — for testing whether a state property currently holds. By state property we mean a combination of fluents formed by the usual first order logical connectives as defined in definition 2.2. These two special predicates are evaluated wrt. the background action domain D. Prior to giving the formal definition of ALPs, let us introduce them via some small examples:

Example 3.1 (Action Logic Program)**.** *In the classic planning domain of the blocks world the following ALP intuitively encodes a strategy for placing every block directly on the table:*

```
strategy :- ?(forall(X,(on(X,table) or X=table))).
strategy :- do(move(Block,X,table)), strategy.
```

The first clause describes the goal: Everything but the table is located directly on the table. The second clause says that our strategy consists of first moving some block to the table, and then to continue searching. The query `?- strategy.` *asks whether the strategy achieves the goal wrt. the background action domain* D *that defines action preconditions and effects.*
The general formulation of planning problems without heuristics as ALPs is as follows:

```
strategy :- ?(goal).
strategy :- do(A), strategy.
```

As is apparent in the examples, ALPs do not contain an explicit notion of time. What is more, they do not by themselves admit a meaningful logical reading. Consider the rule `strategy :- do(A), strategy.` from example 3.1: In a classical logical reading this is just a tautology.

Therefore, the semantics of ALPs is given by macro-expanding them to temporalized definite programs by adding one or two arguments of sort TIME to every program literal, denoting the (possibly identical) start and end of the time interval in which the literal holds. This method of macro-expansion is illustrated by example 3.2. Observe that the special atom `?/1` is expanded to HOLDS, and that the special atom `do/1` is expanded to Poss.

Example 3.2 (Macro-Expanded ALP)**.** *Let* `P` *be the program from example 3.1. Macro-expansion yields the following formulas* P*:*

$$(\forall)\mathit{Strategy}(s,s) \subset \mathit{HOLDS}((\forall x)\mathit{On}(x,\mathit{Table}) \vee x = \mathit{Table}), s)$$
$$(\forall)\mathit{Strategy}(s_1,s_3) \subset \mathit{Poss}(\mathit{Move}(\mathit{block},x,\mathit{Table}),s_1,s_2) \wedge \mathit{Strategy}(s_2,s_3)$$

The query `?- strategy.` *is answered by proving that* $(\exists s)\mathit{Strategy}(S_0,s)$ *is logically entailed by* P *together with* D*, a background axiomatization of the blocks world, e.g., the one given in example 2.1.*

The intuition for the programmer is that the rules in an ALP `P` are sequences of literals where the ordering of the sequence determines the temporal order. These rules are expanded to a set P of classical Horn clauses. For a visible distinction, we typeset unexpanded ALPs in Prolog syntax and use first order syntax for their expansions.

3.1 Syntax of Action Logic Programs

The syntax of ALPs is based on the language of the underlying action domain which we assume to be a sorted logic language including fluents, actions, and objects.

Definition 3.1 (Syntax of ALPs)**.** *The syntax of ALPs over action domains* D *is defined as follows:*

- *The signature of a program* `P` *includes terms corresponding to the terms of sort* OBJECT*,* ACTION *and* FLUENT *of* D*, and also function symbols corresponding to the predicates of the action domain* D *with the exception of the predicates Poss and Holds.*[1] *Additionally it contains "program-only" predicates (obligatory) and terms (optional).*
- *If* `p` *is an n-ary relation symbol from the program signature and* `T1,..,Tn` *are terms then* `p(T1,...,Tn)` *is a program atom.*
- `do(A)` *is a special atom, where* `A` *is an action term.*

[1] All these terms are needed for a term-encoding of state properties.

- `?(Phi)` *is a special atom, where* `Phi` *is a state property.*[2]

- *Rules, programs, and queries are then defined as is usual for pure Prolog programs — only that rule heads must not be special atoms.*

Although ALPs are many-sorted, in the following we will abstract from this situation, assuming that programs and queries are always correctly sorted. We assume that for a programmer it is easy to determine the appropriate sorts for terms.

Next we formally define the macro-expansion of an ALP `P` to its temporalization `P`. The expansion uses two expressions that are defined in the underlying action theory: The predicate $\text{Poss}(a, s, t)$, which means that action a is possible starting at time s and ending at time t; and the macro $\text{HOLDS}(\phi, s)$, meaning that state property ϕ is true at time s. Both have formally been introduced in definition 2.2.

Definition 3.2 (Macro-Expansion of ALPs)**.** *For an ALP rule* `H :- B1, ..., Bn.`*, where* $n \geq 0$*, let* $s_1, \ldots, s_{n+1}$ *be variables of sort* TIME *from the action domain* `D`*.*

- *For* `i=1,...,n`*, if* `Bi` *is of the form*
  - `p(T1,...,Tm)`*, expand it to* $P(t_1, \ldots, t_m, s_i, s_{i+1})$*.*
  - `do(A)`*, expand it to* $Poss(a, s_i, s_{i+1})$*.*
  - `?(Phi)`*, expand it to* $HOLDS(\phi, s_i)$ *and let* $s_{i+1} = s_i$*.*

- *The head atom* `H = p(T1,...,Tm)` *is expanded to* $P(t_1, \ldots, t_m, s_1, s_{n+1})$*.*

- *Finally, we replace* `:-` *by logical implication* $\subset$ *and* `','` *by logical conjunction* $\wedge$ *and take the universal closure of the resulting clause.*

Queries `Q1,...,Qn` *are expanded exactly like rules, only that*

- *the constant* S_0 *— denoting the earliest time-point in* `D` *— takes the place of* s_1*;*

- *we take the existential closure of the resulting clause.*

It is crucial to observe that only by treating the macro $\text{HOLDS}(\phi, s)$ as atomic in an expanded program `P` the latter can indeed be viewed as a set of first order Horn clauses. By expanding `do/1` to Poss we stipulate that actions can only be executed if the underlying action theory entails that the action is executable. The reader is invited to have another look at example 3.2 to see how macro-expansion turns a set of ALP rules into a set of classical first order Horn clauses.

[2] State properties are term-encoded, e.g., `or(on(block2,block1),on(block1,table))`.

3.2 Semantics of Action Logic Programs

One objective in the design of ALPs was to obtain a simple and intuitive declarative semantics. The semantics of ALPs is given by their macro-expansion — a set of first order formulas — together with a first order axiomatization of the action domain. Thus the declarative semantics of ALPs is the usual first order semantics.

3.3 Proof Theory

In this section we introduce two proof calculi for expanded ALPs. These calculi provide the operational semantics of ALPs. Roughly speaking, one of the calculi addresses the general case, while the other is tailored for background theories that enjoy a witness property.

For both proof calculi we stipulate that the background action theory extend the unique name assumption also to terms of sort OBJECT. This requirement is motivated by the fact that otherwise in general we have to perform equational unification between program atoms, blurring the clean separation between program and background theory:

Example 3.3 (Unique Names for Objects)**.** *Assume the action theory entails both* $Holds(F(B), S_0)$ *and* $A = B$*. Consider an ALP that contains only the fact* `p(b).`*, along with the query* `?- ?(f(a)), p(a).`*. Assume that after the derivation step for* $HOLDS(F(A), S_0)$ *we want to apply the standard logic programming derivation rule to the derivation state* $< \neg P(A), \theta >$*: The atoms* $P(A)$ *and* $P(B)$ *can only be unified modulo an equational theory, something which is beyond standard logic programming.*

Let us continue by highlighting the problems that can arise if program atoms and special atoms share variables of sort OBJECT. It is clear that—for a query ϱ—, if there is a substitution θ such that $\mathsf{D} \cup \mathsf{P} \models (\forall)\varrho\theta$ then $\mathsf{D} \cup \mathsf{P} \models (\exists)\varrho$. However, the converse is not true in general. The problems stem from two sources, namely logical disjunction and existential quantification:

Example 3.4 (Disjunction and Existential Quantification)**.** *In the blocks world we might be facing the following state of affairs: We know that there are some blocks on the table, and that one of* $Block_1$ *or* $Block_2$ *is among them; i.e. we have*

$$\mathsf{D}_{\mathsf{Init}} = \{Holds(On(Block_1, Table), S_0) \vee Holds(On(Block_2, Table), S_0)\}.$$

Now for the query $(\exists)\varrho = (\exists x)HOLDS(On(x, Table), S_0)$ *there is no substitution* θ *such that* $\mathsf{D} \models (\forall)\varrho\theta$*.*

Next assume we have the even weaker

$$\mathsf{D}_{\mathsf{Init}} = \{(\exists x)\mathit{Holds}(\mathit{On}(x, \mathit{Table}), S_0)\}.$$

Again, for the query $(\exists x)\mathit{HOLDS}(\mathit{On}(x, \mathit{Table}), S_0)$*, no suitable substitution exists.*

The completeness result for classical logic programming crucially hinges on this converse, namely that whenever $\mathsf{P} \models (\exists)\varrho$ for program P and query ϱ then there is a substitution θ such that $\mathsf{P} \models (\forall)\varrho\theta$. Given that in general the converse does not holds for ALPs in general we cannot give a complete operational semantics via plain SLD-resolution.

In the next section we identify conditions on action domains that make the use of plain SLD-resolution possible and present the resulting proof calculus. In the subsequent section we present a complete proof calculus for the general case.

3.3.1 Elementary Case — LP(D)

A first order theory X is Henkin (or has the witness property) iff. for every sentence of the form $(\exists x)\varphi(x)$ where we have that $\mathsf{X} \vDash (\exists x)\varphi(x)$ then there also is a constant C such that $\mathsf{X} \vDash \varphi(x)[x/C]$ [Enderton, 1972]. As witnessed by, e.g., $(\exists x)\mathrm{Holds}(\mathrm{On}(x, \mathrm{Table}), S_0)$, neither of the two domain axiomatizations from example 3.4 exhibits this property. We define a background theory to be query complete if it is Henkin with regard to all atoms that can occur in ALPs as queries against the background theory:

Definition 3.3 (Query-Completeness)**.** *A background action domain axiomatization* D *is* query-complete *if and only if, for* γ *being* $\mathit{HOLDS}(\phi, s)$ *or* $\mathit{Poss}(a, s_1, s_2)$*, it holds that* $\mathsf{D} \vDash (\exists)\gamma$ *implies that there exists a substitution* θ *such that* $\mathsf{D} \vDash (\forall)\gamma\theta$*.*

In what we call the elementary case we only admit query-complete background theories. Observe however, that in general it is unfortunately not even decidable whether a given action domain D is query-complete. For this we would have to establish whether an arbitrary first order theory (the action domain) is equivalent to a set of Horn clauses.

The proof calculus for the elementary case can now easily be adapted from plain SLD-resolution: for action domain D, expanded ALP P, and query $(\exists)\varrho$, we prove that $\mathsf{D} \cup \mathsf{P} \models (\exists)\varrho$ by proving $\mathsf{D} \cup \mathsf{P} \cup \{(\forall)\neg\varrho\}$ unsatisfiable. The negation of a query $(\exists)G_1 \wedge \ldots \wedge G_n$ is a universally quantified negative clause $(\forall)\neg G_1 \vee \ldots \vee \neg G_n$. The G_i range over all goals, the special atoms that are evaluated against D will be denoted by C. The notions of state, derivation, computed answer substitution are the usual notions from definite logic programs; cf. section 2.2.1. The inference steps $\mathsf{D} \vDash \mathrm{HOLDS}(\phi, s)$ and $\mathsf{D} \vDash \mathrm{Poss}(a, s, s')$ are treated as atomic, assuming the existence of a sound and complete reasoner on D.

Definition 3.4 (Proof Calculus — Elementary Case)**.** *The proof calculus is given by the two rules of inference, one for normal program atoms and one for the special atoms. In particular we have:*

- Program Atoms*:*

$$\frac{< (\neg G_1 \vee \ldots \vee \neg G_i \vee \ldots \vee \neg G_n),\ \theta_1 >}{< (\neg G_1 \vee \ldots \vee \bigvee_{j=1..m} \neg B_j \vee \ldots \vee \neg G_n)\theta_2,\ \theta_1\theta_2 >}$$

  where G_i is a program atom and $(H \subset B_1 \wedge \ldots \wedge B_m)$ is a fresh variant of any clause in P such that G_i and H unify with most general unifier θ_2.

- Special Atoms*:*

$$\frac{< (\neg G_1 \vee \ldots \vee \neg C \vee \ldots \vee \neg G_n),\ \theta_1 >}{< (\neg G_1 \vee \ldots \vee \neg G_n)\theta_2,\ \theta_1\theta_2 >}$$

  where $\mathsf{D} \vDash (\forall)C\theta_2$ with substitution θ_2 on variables in the special atom C.[3]

For expanded action logic program P, background domain axiomatization D and query $(\exists)\varrho$, the following soundness result holds (cf. also theorem 2.1):

Proposition 3.1 (Soundness)**.** *If there exists a successful derivation starting from $< \neg(\exists)\varrho, \epsilon >$ and ending in $< \bot, \theta >$ then $\mathsf{P} \cup \mathsf{D} \vDash (\forall)\varrho\theta$.*

Likewise we obtain the following completeness result, analogous to theorem 2.2 for plain SLD-resolution:

Proposition 3.2 (Completeness)**.** *If $\mathsf{P} \cup \mathsf{D} \vDash (\forall)\varrho\theta_1$, then there exists a successful derivation via any computation rule in our proof calculus starting with $< \neg(\exists)\varrho, \epsilon >$ and ending in $< \bot, \theta_2 >$. Furthermore, there is a substitution θ_3, such that $\varrho\theta_1 = \varrho\theta_2\theta_3$.*

We prove proposition 3.2 by adapting Stärk's proof of the completeness of plain SLD-resolution [Stärk, 1990]. We start by introducing our variant of Stärk's auxiliary concept of an implication tree.

Definition 3.5 (Implication Tree)**.** *A finite tree with atoms as nodes is an* implication tree *wrt. $\mathsf{D} \cup \mathsf{P}$ if for all nodes V:*

- *there is an atom H — i.e. a fact — in P and a substitution θ, such that $V = H\theta$ and V has no children, or*

[3] Observe that, by query-completeness of D, if $\mathsf{D} \vDash \exists C$ then there exists a suitable substitution θ_2.

- *there is a literal $\neg C$ occurring in a clause from P, where C is a special atom, and a substitution θ such that $V = C\theta$ and $\mathsf{D} \models C\theta$ and V has no children, or*
- *there is a clause $H \subset B_1 \wedge \ldots \wedge B_m$ in P $(1 \leq m)$ and a unifier θ, such that $V = H\theta$ and $B_1\theta, \ldots, B_m\theta$ are exactly the children of V.*

Lemma 3.1. *Let $(\exists)G$ be an atom and $\mathsf{D} \cup \mathsf{P} \models (\exists)G$. Then there exists a substitution θ such that $(\forall)G\theta$ has an implication tree wrt. $\mathsf{D} \cup \mathsf{P}$.*

Proof. For special atoms this holds by the definition of query-complete theories. The rest of Stärk's proof of this lemma goes through unchanged.⊣ □

Stärk's actual completeness proof based on this lemma does not require any modification.

Given the independence from the computation rule, we adopt the standard leftmost rule from Prolog. This ensures that the first argument of sort TIME of any atom will be instantiated prior to evaluation, whereas the second one will be thereafter.

Observe that the proof calculus for the elementary case can only be used to solve unconditional planning problems, where "unconditional" means that there always exists a single sequence of actions achieving the goal, and not a disjunction of sequences of actions.

It is also worth pointing out that our notion of query-completeness is orthogonal to the following more common notion of completeness: A first order theory is called complete iff. for every sentence φ either φ or $\neg\varphi$ is in the theory.

Example 3.5 (Completeness vs. Query-Completeness)**.** *For example, if in our blocks world example 3.4 we do not include any information whatsoever concerning the location of blocks, the resulting theory is query-complete but not complete.*

On the other hand, the following theory is complete but not query-complete: For the sake of the argument, assume our action domain does not contain actions or fluents. Let there be a single predicate P, and a single constant A. We axiomatize that there are only two things, one is denoted by A and the other is an unnamed object y such that $P(y)$:

$$(\exists x)(\exists y)x \neq y \wedge (\forall z)(z = x \vee z = y) \wedge x = A \wedge \neg P(A) \wedge P(y)$$

This complete theory entails $\exists x P(x)$; but there is not a suitable substitution θ such that $\forall P(y)\theta$ is entailed, and, hence, this theory is not query-complete.

Query-complete action domains cannot, however, represent disjunctive or purely existential information concerning terms of sort OBJECT — this lack of expressivity is their very purpose. But arguably one of the strong-points of general action calculi is that this kind of information can naturally be represented. Next we move to this more interesting setting.

3.3.2 General Case — CLP(D)

We address the problems illustrated by example 3.4 by moving to the richer framework of constraint logic programming [Jaffar and Lassez, 1987]. By using this framework we will be able to define a sound and complete proof calculus that can cope with disjunctive and merely existential information.

Constraint logic programming — CLP(X) — constitutes a family of languages, parametrized by a first order constraint domain axiomatization X. In addition to the ordinary atoms of plain logic programming in CLP(X) there are special atoms — the constraints — that are evaluated against the background constraint theory X. A concise summary of the framework is contained in section 2.2.2.

We instantiate CLP(X) to CLP(D) — constraint logic programming over action domains D — taking as constraint atoms C the special atoms $\mathrm{Poss}(a, s_1, s_2)$ and $\mathrm{HOLDS}(\phi, s)$.

Disjunctive Substitutions

As illustrated by example 3.4, in the case of non-query-complete domains there need not be unique most general substitutions — in the example we do not know which of the two blocks is on the table. However, we have the disjunctive information that one of the two blocks is on the table — something that leads us to resorting to the notion of (most general) disjunctive substitutions, formally defined as follows:

Definition 3.6 (Disjunctive Substitution)**.** *A disjunctive substitution is a finite set of substitutions* $\Theta = \{\theta_1, \ldots, \theta_n\}$*. Applying the disjunctive substitution* Θ *to a clause* φ *results in the disjunction* $\bigvee_{i=1..n} \varphi\theta_i$*. Given two disjunctive substitutions* Θ_1, Θ_2 *their composition* $\Theta_1\Theta_2$ *is defined as* $\{\theta_i\theta_j \mid \theta_i \in \Theta_1 \text{ and } \theta_j \in \Theta_2\}$*. A disjunctive substitution* Θ_1 *is more general than a disjunctive substitution* Θ_2 *if for every* $\theta_i \in \Theta_1$ *there exist* $\theta_j \in \Theta_2$ *and* θ *such that* $\theta_i\theta = \theta_j$*.*

To every disjunctive substitution Θ there corresponds a formula in disjunctive normal form consisting only of equality atoms. With a little abuse of notation we will denote this formula as Θ, too; e.g., we treat $\{\{x \to \mathrm{Block}_1\}, \{x \to \mathrm{Block}_2\}\}$ and $x = \mathrm{Block}_1 \vee x = \mathrm{Block}_2$ both as the disjunctive answer for example 3.4.

General proof calculi that can compute most general disjunctive answers for arbitrary first order theories are, e.g., full resolution with the help of answer literals [Green, 1969] or restart model elimination [Baumgartner et al., 1997] — with the caveat of differentiating between terms of sort OBJECT and auxiliary Skolem functions.

The General Proof Calculus

In CLP(X), the derivation rule for the constraint atoms C is based on the logical equivalence (wrt. X) of $C \wedge \sigma$ and σ', where σ is the constraint store prior to rule application, and σ' the resulting constraint store. More precisely, it is based on the formula $\mathsf{X} \vDash (C \wedge \sigma) \equiv \sigma'$. In our case σ' can be obtained by exploiting the following two logical equivalences:

- If there is a substitution Θ such that $\mathsf{D} \vDash (\forall) \bigvee_{\theta \in \Theta} C\theta$ then we can exploit that
$$\mathsf{D} \vDash (C \wedge \sigma) \equiv ((C \wedge \Theta) \wedge \sigma). \tag{3.1}$$
- Otherwise we use the weaker fact that
$$\mathsf{D} \vDash C \wedge \sigma \equiv C \wedge \sigma. \tag{3.2}$$

For the sake of efficiency the CLP(X) framework is usually augmented by "Solve"-transitions [Frühwirth and Abdennadher, 2003]. These replace states in a derivation by equivalent simpler ones, foremost by rewriting the constraint store. For example, applying (non-disjunctive) substitutions is a "Solve"-transition, where, e.g., the state $< \neg P(x), x = 1 >$ is rewritten to the equivalent $< \neg P(1), \top >$. Applying disjunctive substitutions is not as straightforward:

Example 3.6. *Consider the query* `?- ?(on(X,table)), p(X).` *on top of the disjunctive action domain* D *from example 3.4. Further assume that the ALP contains the clause* `p(X) :- ?(on(X,table)).`*, chosen for the sake of illustration. The derivation starts with state*

$$< \neg HOLDS(On(x, Table), S_0) \vee \neg P(x, S_0), \top > .$$

After deriving $HOLDS(On(x, Table), S_0)$ *— using 3.1 — we obtain the state*

$$< \neg P(x, S_0),\ (HOLDS(On(x, Table), S_0) \wedge \Theta) >,$$

where Θ *is the formula* $x = Block_1 \vee x = Block_2$*. Next, by applying the substitution* Θ *to* $P(x, S_0)$*, we obtain*

$$< \neg P(Block_1, S_0) \vee \neg P(Block_2, S_0),\ HOLDS(On(x, Table), S_0) \wedge \Theta >,$$

which reduces to

$$< \neg HOLDS(On(Block_1, Table), S_0) \vee \neg HOLDS(On(Block_2, Table), S_0),$$
$$HOLDS(On(x, Table), S_0) \wedge \Theta > .$$

But now we cannot proceed by querying whether

$$\mathsf{D} \cup \mathsf{P} \models \mathit{HOLDS}(\mathit{On}(\mathit{Block}_1, \mathit{Table}), S_0)$$

or

$$\mathsf{D} \cup \mathsf{P} \models \mathit{HOLDS}(\mathit{On}(\mathit{Block}_2, \mathit{Table}), S_0).$$

However, it is obvious that the query

$$\mathsf{D} \cup \mathsf{P} \models \mathit{HOLDS}(\mathit{On}(\mathit{Block}_1, \mathit{Table}), S_0) \vee \mathit{HOLDS}(\mathit{On}(\mathit{Block}_2, \mathit{Table}), S_0),$$

admits a successful derivation. But, for simplicity, it would be nice to obtain a proof calculus that operates on single literals. In the example this can be achieved by first splitting the state into a disjunction of substates, one for each of the simple substitutions $\theta_1, \theta_2 \in \Theta$*:*

$$\begin{aligned} &< \neg P(\mathit{Block}_1, S_0),\ \mathit{HOLDS}(\mathit{On}(\mathit{Block}_1, \mathit{Table}), S_0) > \\ \dot{\vee}\ &< \neg P(\mathit{Block}_2, S_0),\ \mathit{HOLDS}(\mathit{On}(\mathit{Block}_2, \mathit{Table}), S_0) > . \end{aligned}$$

Considering, e.g., the first disjunct we observe that still

$$\mathsf{D} \not\models \mathit{HOLDS}(\mathit{On}(\mathit{Block}_1, \mathit{Table}), S_0);$$

if, however, we augment the action domain by the corresponding case, we are successful:

$$\mathsf{D} \cup \{\mathit{HOLDS}(\mathit{On}(\mathit{Block}_1, \mathit{Table}), S_0)\} \models \mathit{HOLDS}(\mathit{On}(\mathit{Block}_1, \mathit{Table}), S_0).$$

Generalizing this idea, in our proof calculus we employ reasoning by cases, whenever we have obtained a disjunctive substitution $\Theta = \{\theta_1, \ldots, \theta_n\}$: We split the current substates of the derivation into a disjunction of substates, one for each θ_i, and extend the simple substates from pairs to triples, adding an additional argument ζ for recording the assumed case. Then in each substate $<$ Negative Clause$, \sigma, \zeta >$ special atoms are evaluated against the action domain augmented by the corresponding cases: $\mathsf{D} \cup \{\zeta\}$.

It is crucial to observe that the disjunction of all the newly introduced substates on one level of the derivation tree is equivalent to the original state. In particular the disjunction of all the domain axiomatizations D' augmented by the respective cases ζ is equivalent to the original D.

A state now is a disjunction of (simple) substates $<$ Negative Clause$, \sigma, \zeta >$, and the symbol used for denoting disjunctions of substates is $\dot{\vee}$. A derivation of the query ϱ starts with the simple state $< \neg\varrho, \top, \top >$. A derivation is successful if it is ending in $\dot{\bigvee}_{i=1..n} < \bot, \sigma_i, \zeta_i >$. The formula $\bigvee \sigma_i$ is the computed answer. We will define our proof calculus to operate on individual literals in single substates.

Any derived substate that is different from $< \bot, \sigma_i, \zeta_i >$ and to which none of the reduction rules can be applied indicates a failed derivation.

Formally, the proof calculus is given by the rules of inference depicted in figure 3.1. We omit the straightforward rule of inference for program atoms.

Substitution Rule:

$$\frac{< (\neg G_1 \vee \ldots \vee \neg C \vee \ldots \vee \neg G_n), \sigma, \zeta >}{< \dot{\bigvee}_{i=1..k} < (\neg G_1 \vee \ldots \vee \neg G_n), \sigma \wedge C \wedge \theta_i, \zeta \wedge C \wedge \theta_i >}$$

where $\mathsf{D} \cup \{\zeta\} \models (\forall) \bigvee_{\theta_i \in \Theta} C\theta_i$ with most general disjunctive substitution Θ

Constraint Rule:

$$\frac{< (\neg G_1 \vee \ldots \vee \neg C \vee \ldots \vee \neg G_n), \sigma, \zeta >}{< (\neg G_1 \vee \ldots \vee \neg G_n), \sigma \wedge C, \zeta >}$$

if $\mathsf{D} \cup \{\zeta\} \not\models \neg(\exists)\ \sigma \wedge C$.

Figure 3.1: Rules of Inference for CLP(D)

Piggybacking on the results for the CLP(X) scheme we obtain both soundness and completeness results for CLP(D); for program P, query ϱ, and domain axiomatization D we have:

Theorem 3.1 (Soundness of CLP(D))**.** *If ϱ has a successful derivation with computed answer $\bigvee_{i=1..k} \sigma_i$ then $\mathsf{P} \cup \mathsf{D} \models (\forall) \bigvee_{i=1..k} \sigma_i \supset \varrho$.*

Theorem 3.2 (Completeness of CLP(D))**.** *If $\mathsf{P} \cup \mathsf{D} \models (\forall)\sigma \supset G$ and σ is satisfiable wrt. D, then there are successful derivations for the goal G with computed answers $\sigma_1, \ldots, \sigma_n$ such that $\mathsf{D} \models (\forall)(\sigma \supset (\sigma_1 \vee \ldots \vee \sigma_n))$.*

Let us illustrate the power of CLP(D) by the following example; it highlights the capability of CLP(D) to infer disjunctive plans:

Example 3.7 (Disjunctive Plans)**.** *Assume the initial state is specified by*

$$(Holds(On(Block_1, Table), S_0) \vee Holds(On(Block_2, Table), S_0)) \wedge$$
$$\neg(\exists x) Holds(On(x, Block_1), S_0) \wedge \neg(\exists x) Holds(On(x, Block_2), S_0) \wedge$$
$$Holds(On(Block_3, Table), S_0) \wedge \neg(\exists x) Holds(On(x, Block_3), S_0);$$

so we know that $Block_1$ or $Block_2$ are on the table with no obstructing block on top of them, just as $Block_3$ is. Let the precondition and the effects of moving a block be

axiomatized as in example 2.1:

$$
\begin{aligned}
&(\forall) Poss(Move(block_1, x, y), s_1, s_2) \equiv \\
&\qquad Holds(On(block_1, x), s_1) \wedge x \neq y \wedge \\
&\qquad (\neg\exists block_2) Holds(On(block_2, block_1), s_1) \wedge \\
&\qquad (\neg\exists block_3)(Holds(On(block_3, y), s_1) \vee y = Table) \wedge \\
&\qquad s_2 = Do(Move(block_1, x, y), s_1) \\
&(\forall) Poss(Move(block, x, y), s_1, s_2) \supset \\
&\quad [(\forall f)(f = On(block, y) \vee (Holds(f, s_1) \wedge f \neq On(block, x))) \equiv Holds(f, s_2)].
\end{aligned}
$$

Consider an ALP goal `?- do(move(X,block3)).`*. For this we obtain a successful derivation containing two substates, informing us that we can achieve our goal by the disjunctive plan consisting of*

- *moving* $Block_1$ *atop* $Block_3$*, or*
- *moving* $Block_2$ *atop* $Block_3$*.*

However, it is important to note that this disjunctive plan does not yet tell us which of the two alternatives will succeed. All it does is tell us that we can achieve the goal if we can find out which of the two alternatives actually holds. This issue, and ways around it, will be discussed in more detail below in section 3.6.2.

The Importance of Domain Closure on Actions

Let us next illustrate another peculiarity of the CLP(D) proof calculus:

Example 3.8 (The Universal Plan)**.** *Consider the following program, where* `Phi` *shall denote the ALP-encoding of an arbitrary planning goal:*

```
succeed :- do(A), ?(Phi).
```

*Via the Constraint Rule of the CLP(*D*) proof calculus we obtain a successful derivation with constraint store* $Poss(a, S_0, s) \wedge HOLDS(\phi, s)$*. This informs us that we can achieve our goal* ϕ *if there exists an hitherto unspecified action* a *such that* ϕ *holds after the execution of* a*.*

Of course, such an answer is not very helpful, and the above example clearly indicates the need for a domain closure axiom on actions in D: Let $\mathfrak{A}$ denote the set of all functions into sort Action in D. Then the following is the desired domain closure axiom on actions:

$$(\forall a, \vec{x}) \bigvee_{A \in \mathfrak{A}} a = A(\vec{x}).$$

Henceforth, we stipulate that all D contain such an axiom.

3.3.3 Refinements and Extensions of the Proof Calculi

We proceed by detailing two refinements of the CLP(D) proof calculus: We first address the question how to avoid conditional answers — at least sometimes. Then we discuss an even more general notion of answer (substitutions) — in some cases it may be considered useful to also admit inequality constraints and even more general answer constraints. We then add a brief discussion of how both the LP(D) and the CLP(D) proof calculus can be extended by negation as finite failure.

Conditional Answers in CLP(D)

As already pointed out, in constraint logic programming the answers in CLP(D) are conditional: for an answer $\bigvee_{i=1..k} \sigma_i$ to a query we have to check whether $\mathsf{D} \models \bigvee_{i=1..k} \sigma_i$ to see whether the query is indeed entailed. We observe that in CLP(D) each single constraint store σ_i is a logical conjunction; thus σ_i is entailed by D iff. every individual constraint C in σ_i is entailed.

For those constraints that were added using equivalence 3.1 by the end of a derivation this has already been established, hence these constraints don't have to be added to the constraint store. The status of those constraints that were added using equivalence 3.2 is not clear, though — they may be entailed by D, or they may only be consistent with D. Accordingly, we propose to introduce two versions of the constraint rule (cf. figure 3.1) by distinguishing the two logical possibilities that

- either it is the case that $\mathsf{D} \cup \{\zeta\} \models (\exists)\ \sigma \wedge C$, or
- it holds that $\mathsf{D} \cup \{\zeta\} \not\models (\exists)\ \sigma \wedge C$ and $\mathsf{D} \cup \{\zeta\} \not\models \neg(\exists)\ \sigma \wedge C$. [4]

Of course, for a constraint C that is added to the constraint store by either of these two variants of the constraint rule, at a later stage of the derivation we cannot tell by which variant of the rule it was added. Hence we extend the calculus by a marking mechanism, for recording which constraints are already solved. If a constraint C is added to the store σ using that $\mathsf{D} \models (\exists)\ \sigma \wedge C$, all constraints in the store are marked as solved. However, the second of the above two logical possibilities may require that a constraint marked as solved must be marked as unsolved again, as highlighted by the following example:

Example 3.9 ((Un-)Marking Constraints as Solved)**.** *Assume we want to answer the query* `?- ?(on(Block,table)), ?(on(Block,block1).`*, where the special atoms share the variable* `Block`*. Further assume that the initial state is axiomatized as*

$$(\exists x)\mathit{Holds}(\mathit{On}(x, \mathit{Table}), S_0)$$
$$(\exists x)\mathit{Holds}(\mathit{On}(x, \mathit{Block}_1), S_0).$$

[4] The final version of the two constraint rules is only given in figure 3.2 below.

Clearly, $\mathsf{D} \models (\exists x) Holds(On(x, Table), S_0)$ *and* $\mathsf{D} \models (\exists x) Holds(On(x, Block_1), S_0)$*, but* $\mathsf{D} \not\models (\exists x)\ Holds(On(x, Table), S_0) \wedge Holds(On(x, Block_1), S_0)$*. In the first step of a derivation we establish that the special atom is entailed, add it to the constraint store, and mark it as solved. In the next derivation step, however, it becomes clear that the constraint has to be marked as unsolved again.*

Non-ground constraints that are marked as solved at the end of a derivation can be omitted from the answer constraint. Refining the Mark() and Unmark() operations is therefore crucial: More fine-grained marking operations will also allow us to omit more ground constraints from the constraint store. For this purpose we introduce the notion of relevant constraint:

Definition 3.7 (Relevant Constraint)**.** *Let* σ *be the constraint store prior to rule application, and let* C *be the selected constraint. A constraint* C_i *occurring in* σ *is relevant for* C *if and only if it either*

- *shares variables with* C*; or*
- *shares variables with another relevant constraint.*

Based on this definition we observe that in a derivation state where C is the selected constraint we have the following:

- If we check that $\mathsf{D} \cup \{\zeta\} \models (\exists) \bigwedge_i C_i \wedge C$ we only consider relevant constraints C_i. Of course, we also only mark the C_i and C as solved.
- If we establish that $\mathsf{D} \cup \{\zeta\} \not\models (\exists)\ \sigma \wedge C$ and $\mathsf{D} \cup \{\zeta\} \not\models \neg(\exists)\ \sigma \wedge C$ we call the Unmark() operation only on the constraints relevant for C, too.

The refined rules of inference for CLP(D) are depicted in figure 3.2; the operations Mark() and Unmark() take formulas from the constraint store as argument and annotate the occurring relevant literals as solved or unsolved, respectively.

As before, a derivation is successful if it is ending in $\dot{\bigvee}_{i=1..n} < \bot, \sigma_i, \zeta_i >$. But now the formula $\bigvee \sigma_i^*$ is the computed answer, where σ_i^* is obtained from σ_i by omitting any constraint marked as solved. For this refined proof calculus and notion of computed answer our soundness and completeness results from theorems 3.1 and 3.2 continue to hold.

Obviously, Constraint Rule I can be applied in cases where the Substitution Rule is applicable, too. However, the latter yields a higher information content. Hence, for an actual implementation of the CLP(D) proof calculus it is reasonable to adopt the following strategy:

- first we check for the existence of a (disjunctive) substitution; and
- next we check for the (existential) entailment of the constraint C; and

Substitution Rule:

$$\frac{< (\neg G_1 \vee \ldots \vee \neg C \vee \ldots \vee \neg G_n), \sigma, \zeta >}{\dot{\bigvee}_{i=1..k} < (\neg G_1 \vee \ldots \vee \neg G_n), \sigma, \zeta \wedge C \wedge \theta_i >}$$

where $\mathsf{D} \cup \{\zeta\} \models (\forall) C \wedge \Theta$ with most general disjunctive substitution $\Theta = \bigvee_{i=1..k} \theta_i$

Constraint Rule I:

$$\frac{< (\neg G_1 \vee \ldots \vee \neg C \vee \ldots \vee \neg G_n), \sigma, \zeta >}{< (\neg G_1 \vee \ldots \vee \neg G_n), \mathsf{Mark}(\sigma \wedge C), \zeta >}$$

if $\mathsf{D} \cup \{\zeta\} \models (\exists) \sigma \wedge C$.

Constraint Rule II:

$$\frac{< (\neg G_1 \vee \ldots \vee \neg C \vee \ldots \vee \neg G_n), \sigma, \zeta >}{< (\neg G_1 \vee \ldots \vee \neg G_n), \mathsf{Unmark}(\sigma \wedge C), \zeta >}$$

if $\mathsf{D} \cup \{\zeta\} \not\models (\exists) \sigma \wedge \neg C$ and $\mathsf{D} \cup \{\zeta\} \not\models \neg(\exists) \sigma \wedge C$.

Figure 3.2: Refined Rules of Inference for CLP(D)

- only then we try to verify that the constraint C is at least consistent with D and P.

To gain a better understanding of the role played by conditional answers for the completeness of CLP(D) consider the following example:

Example 3.10 (Conditional Answers in CLP(D))**.** *Assume that an ALP* P *is defined by the following clauses over an empty domain description* D*:*

```
a :- ?(Phi).
a :- ?(not Phi).
```

— where `Phi` *shall denote some arbitrary state property. Clearly* $\mathsf{D} \cup \mathsf{P} \models A$ *because of* $\models \phi \vee \neg\phi$*. In* P*, A has two derivations, one ending with the constraint store* ϕ*, and an analogous one ending with the store* $\neg\phi$*.*

This example illustrates the nature of our completeness result for CLP(D): in order to establish that some query ϱ is indeed entailed by a program together with a domain axiomatization it may be necessary to have multiple successful derivations, each ending with constraint store σ_i. The query can then be proven by establishing that $\bigvee_i \sigma_i \models \varrho$. From an implementation point of view this clearly is bad news.

We can only hope that no programmer will feel tempted to use such tautological constructs for her ALP.

Example 3.10. (continued) *Next assume we consider only a single successful derivation, ending, e.g., with constraint store $\sigma = \phi$. This informs the programmer that $\mathsf{P} \cup \mathsf{D} \models \phi \supset A$, meaning that, if she can somehow ensure ϕ to hold, A will also hold.*

From an implementation point of view it is clearly easer to only consider single derivations, informing the programmer that her goal can be achieved, provided that the constraints from the store can be met. This shifts most of the burden on the programmer, but avoids a daunting reasoning task. This very course of action is also taken by virtually all existing implemented CLP(X)-systems.

Constraint Store Handling

We next make two observations that can be exploited to obtain a simpler constraint store:

- Ground constraints, that are marked as solved, can safely be omitted from the constraint store.
- If, for action domain D, we have that $\mathsf{D} \models \sigma \equiv \top$, the constraint store σ obviously can be replaced by $\top$.

The latter of these two observations will probably not prove to be very fruitful in practice: The check is expensive, and not likely to succeed often. The case where $\mathsf{D} \models \sigma \equiv \bot$ is of far greater practical importance, indicating a failed derivation. Note that this case is already addressed by the rules of inference.

Generalized Answer Notions

Strictly speaking, disjunctive substitutions are not necessary for CLP(D); we could also stick with ordinary substitutions, and treat goals as answer constraints if there are no suitable substitutions. Note that this would not at all affect our soundness and completeness results; the computed answers would simply be less informative.

In fact, the Constraint Rule from the proof calculus from figure 3.1 already without the Substitution Rule constitutes a sound and complete proof calculus. In this case for every query ϱ we obtain the computed answer that the query holds if it holds, $\varrho \supset \varrho$. This certainly is the least informative type of answer one can imagine.

On the other hand, the notion of a disjunctive answer substitution is not yet strong enough to generate answers that are maximally informative in all cases, something that can easily be seen in the following example:

Example 3.11 (Problems with Disjunctive Substitutions)**.** *Assume that initially we only know that no block except* $Block_4$ *is directly on the table; i.e. we have:*

$$Holds(On(Block_4, Table), S_0)$$
$$(\forall x)\ x \neq Block_4 \supset \neg Holds(On(x, Table), S_0)$$

The query `?- ?(not(on(X,table))).` *has a successful derivation, informing the programmer that there is something, that is not directly on the table. Albeit it would be more informative if we could also tell the programmer that this something is not* $Block_4$*. A related issue surfaces if we assume that initially we only know that* $Block_2$ *is on the table, unless the table has been toppled:*

$$\neg Holds(Toppled(Table), S_0) \supset Holds(On(Block_2, Table), S_0)$$

A programmer is probably much more happy if the computed answer to her query `?- ?(not(on(X,table))).` *tells her that the query can be satisfied if the table has not been toppled, than if she is only being told that there is something on the table if there is something on the table.*

Based on these considerations we can generalize the notion of answer to an arbitrary first order formula. Both soundness and completeness results for CLP(D) carry over to this setting, if we adapt the derivation rules to the employed notion of answer. A recent survey and study of such a generalized answer notion in the framework of resolution theorem proving can be found in [Burhans and Shapiro, 2007].

However, for this thesis we adopt the view that the notion of answer should be extensional: We consider the generation of (possibly a disjunction of) witnesses for state properties to be the most interesting type of answer, and have defined the CLP(D) proof calculus accordingly. As highlighted by the example 3.11 above this viewpoint is debatable; in any case our proof calculus is flexible enough to accommodate more general, intensional notions of answer.

On Time Structures

Let us at this point digress to issues related to the axiomatization of the underlying time structure — these will be relevant for the next section on negation as finite failure.

Regarding the underlying time structure, we first observe that there are many action domain axiomatizations in the literature that do not include an explicit axiomatization of time. For example, none of the Fluent Calculus action domains in [Thielscher, 2005d] features an axiomatization of situations. Likewise, it has been proved in [Pirri and Reiter, 1999] that Golog programs can safely be evaluated without ever referring to the foundational axioms of the Situation Calculus. For proving

ordinary ALP queries — quantifying existentially over time points — an axiomatization of the time structure is not needed either.

However, an axiomatization of the time structure is needed in order to define certain properties of action domains, or to infer, e.g., domain constraints. For the UAC it has been stipulated that action domains contain a time structure axiomatization that is at least partially ordered, and contains a least element. For practical purposes the following three time structures seem to suffice: The natural numbers, the positive real numbers, and situations. If we want to stick to the first order semantics of ALPs we have to stipulate that these be axiomatized in first order logic, too.

For the first order axiomatization of situations we start from the second order axiomatization given in [Reiter, 2001a]. The only change is that we replace the second order induction axiom by the axiom scheme

$$\phi[S_0] \wedge (\forall s, a)(\phi[s] \supset \phi[\mathrm{Do}(a, s)]) \supset (\forall s')\phi[s'],$$

where ϕ ranges over all state formulas with s the only free variable.

For the natural numbers we can use first order versions of either Presburger or Peano arithmetic; for the real numbers there is Tarski's axiomatization [Shoenfield, 1967]. These time structure axiomatizations are likewise to be augmented by the induction axiom scheme

$$\phi[0] \wedge (\forall s)(\phi[s] \supset \phi[s+1]) \supset (\forall s')\phi[s'].$$

The purpose of these induction schemes is to ensure that it is possible to infer that some property holds at every time point, i.e. proving domain constraints. If we are not using second order logic for the induction axioms this means that the time structure admits non-standard models. In some contexts this may be undesirable — cf. the discussion on planning completeness in section 3.5.2 below. But wrt. domain constraints it seems to be sufficient if we can prove all first order definable invariant state properties.

Negation as Finite Failure

We have defined action logic programs as a collection of Horn clauses. This means that no negated atoms (normal or special) may occur in the rule body.

The logic programming community has long felt the need to also allow negated atoms in the rule body of logic program clauses. For plain logic programs this extension has been achieved by the Clark completion of a logic program [Clark, 1978], where one reads rules as logical equivalences instead of logical implications, and resorts to the computation rule of negation as finite failure.

For ALPs, we can also allow negative literals in rule bodies and evaluate them by the inference rule of negation as finite failure: The respective soundness and restricted completeness results for the constraint logic programming scheme CLP(X)

(cf. [Jaffar et al., 1998, Frühwirth and Abdennadher, 2003]) then naturally extend to ALPs with negation.

But first we have to clarify what a negated goal in an unexpanded ALP should mean. For ordinary program atoms that do not depend on the special atoms the meaning is intuitively quite clear. Negated `?(Phi)` atoms are to be read as $\mathsf{D} \models \neg\text{HOLDS}(\phi, s)$, and negated `do(A)` atoms as $\mathsf{D} \models \neg\text{Poss}(\phi, s)$. In both cases negation as failure does not yet augment the expressivity of the language. But in general, there is an interesting reading of a negated program atom that does depend on the special atoms as a goal being unachievable.

To reflect this reading the macro-expansion of ALPs has to be adapted. In expanded ALPs the goals are no longer directly linked by time variables — rather, for negated goals the second time variable is not part of the chain. Let us illustrate the underlying idea by a small example:

Example 3.12 (Expanding Programs with NaF)**.** *The following schematic program clause is meant to express that in our strategy, if we cannot exploit certain crucial subplans, then we resort to "plan B":*

```
strategy :- \+ sub_plan1, \+ sub_plan2, planB.
```

Here `\+` *denotes negation as finite failure. This is naturally expanded to*

$$(\forall)\mathit{Strategy}(s_1, s_2) \subset \\ \neg(\exists s_3)\mathit{SubPlan}_1(s_1, s_3) \wedge \neg(\exists s_4)\mathit{SubPlan}_1(s_1, s_4) \wedge \mathit{PlanB}(s_1, s_2).$$

The Clark completion can then be applied to expanded programs. As is clear from this example, positive goals are only linked to positive goals.

The Role of Domain Constraints Note that this extension of CLP(D) by negation as finite failure leads to intricate issues. In particular, it is then necessary to prove domain constraints in order to derive ALP queries. Consider the generic encoding of a planning problem as an ALP:

```
strategy :- ?(Phi).
strategy :- do(A), strategy.
```

The query `?- \+ strategy` is expanded to $\neg(\exists s)\text{Strategy}(S_0, s)$, and, hence, can be derived if and only if there is no sequence of actions achieving the goal. In general however, this cannot be proved by checking all (possibly infinite) sequences of actions — as the simple CLP(D) calculus would do. But it may be possible to prove this by considering the program and the underlying action theory as a whole — if the underlying action theory contains a suitable axiomatization of the time structure it may in fact be possible to prove that $\mathsf{D} \models \forall s(\neg\text{HOLDS}(\phi, s))$.

The best we can achieve for our proof calculi, is to say that negation as finite failure is complete if the negated goals are finitely refutable — this is exactly the restricted completeness result for CLP(X) from [Jaffar et al., 1998].

We do not give a proof calculus that can prove domain constraints here. Such a calculus would have to operate on the program as a whole together with the action theory; and, arguably, a less simple proof calculus results in a less usable programming language.

Relation of CLP(D) to CLP(X)

Let us conclude this section by some remarks on how constraint logic programming over action theories (CLP(D)) relates to other instances of the general constraint logic programming scheme CLP(X).

In one direction, the goal is to take advantage of existing implementations of CLP(X) for specific constraint theories X, like, e.g., the natural numbers $\mathbb{N}$, the rational or real numbers $\mathbb{Q}$ and $\mathbb{R}$, finite domain constraints, or set constraints. Recall that the only requirement of the CLP(X) calculus is that the respective constraint theory X is axiomatized in first order logic. If we include the constraint theory X in the foundational axioms $\mathsf{D}_{\mathsf{aux}}$ of D, we can pose ALP queries `?(Phi)`, where the state property ϕ is evaluated against X. This observation provides the theoretical foundation for the integration of existing constraint solvers into an implementation of our ALP framework.

From the opposite perspective, we observe that neither the LP(D) nor the CLP(D) proof calculus is limited to action theories as background theories. LP(D) can serve as a sound and complete proof calculus for any constraint theory that has the witness property (is query-complete). And CLP(D) constitutes a sound and complete proof calculus for those constraint theories that do not exhibit this property. The techniques used for evaluating the Poss and HOLDS atoms can be seen as analogous to existing constraint solving techniques for other constraint theories X.

3.4 Computed Answers and Inferred Plans

Somewhat surprisingly, there remains the question how the inferred plans should be communicated to the programmer. The usual notion of a computed answer for a query in logic programming will not provide any information concerning the inferred plans to the programmer.

The astute reader may have noticed that the variables of sort TIME in computed answers do not occur in unexpanded action logic programs. In the case of situation-based background theories D it is sufficient to print out the final situations. In general this does not suffice, though: neither the natural nor the real numbers serving as time structures provide any information on the sequence of actions performed.

The programmer of an ALP is most interested in plans that achieve goals; that is, she is most interested in the sequence of the evaluated $\text{Poss}(a, s_1, s_2)$ atoms, or, in the case of disjunctive plans, in the corresponding tree. Rather than have the programmer construct this information herself, we can also add this functionality to our proof calculi: In the general case of CLP(D) the constraint stores already contain the necessary information; in the special case of query complete domains the calculus is easily extended to construct this sequence. We have refrained from making this extension explicit in the previous sections in order to keep the presentation simple.

3.5 Planning Completeness

The soundness and completeness results from the previous section assure us that a query can be proved if and only if the ALP together with the action theory entail the query. In this section we consider the following question: Assume that the action theory entails that a goal is achievable, i.e. $\mathsf{D} \models (\exists s)\text{HOLDS}(\phi, s)$ for goal description ϕ. Is there an ALP that can be used to infer a plan achieving the goal? It turns out that, unfortunately, in general there need not be such a program. We next identify a range of properties that may be exhibited by action theories. We then discuss which of these properties are needed to ensure planning completeness. Probably the most drastic measure we have to take to this end is that we use second order axiomatizations of the underlying time structure.

3.5.1 Properties of Background Theories

We proceed by identifying a number of properties that may be exhibited by domain axiomatizations. Not all of these notions will be required for showing planning completeness; it is just convenient to collect all these properties in one definition. Except for the notions of deterministic domains, temporal determinacy, symmetric, universal and anytime domains these notions have already been introduced in [Thielscher, 2007]. Following the definition we give motivating examples, illustrating the introduced notions.

Definition 3.8 (Properties of Action Domains)**.** *An action domain* D *with precondition axioms* $\mathsf{D}_{\mathsf{Poss}}$ *and foundational axioms* $\mathsf{D}_{\mathsf{aux}}$ *is*

- progressing *if*
  - $\mathsf{D}_{\mathsf{aux}} \models (\exists s_1 \forall s_2)\, s_1 \leq s_2$ *and*
  - $\mathsf{D}_{\mathsf{Poss}} \cup \mathsf{D}_{\mathsf{aux}} \models \mathit{Poss}(a, s_1, s_2) \supset s_1 < s_2$.
- sequential *if it is progressing and no two actions overlap; that is*

$$\mathsf{D}_{\mathsf{Poss}} \cup \mathsf{D}_{\mathsf{aux}} \models \mathit{Poss}(a, s_1, s_2) \wedge \mathit{Poss}(a', s_1', s_2') \supset (s_2 < s_2' \supset s_2 \leq s_1') \wedge (s_2 = s_2' \supset a = a' \wedge s_1 = s_1').$$

- ramification free *if for every effect axiom*

$$Poss(A(\bar{x}), s_1, s_2) \supset \eta_1[s_1, s_2] \vee \ldots \vee \eta_k[s_1, s_2]$$

each $\eta_i[s_1, s_2]$ *is of the form*

$$\begin{array}{ll}(\exists \bar{y}_i)(\phi_i[s_1] & \wedge\ (\forall f)\,[\delta_i^+[s_1] \supset Holds(f, s_2)] \\ & \wedge\ (\forall f)\,[\delta_i^-[s_1] \supset \neg Holds(f, s_2)]),\end{array}$$

i.e. s_2 *does not occur in* δ_i^+ *and* δ_i^-*.*

- symmetric *if for every effect axiom*

$$Poss(A(\bar{x}), s_1, s_2) \supset \eta_1[s_1, s_2] \vee \ldots \vee \eta_k[s_1, s_2]$$

each $\eta_i[s_1, s_2]$ *is of the form*

$$(\exists \bar{y}_i)\phi_i[s_1] \quad \wedge\ (\forall f)\,[\delta_i[s_1, s_2] \equiv Holds(f, s_2)],$$

i.e. the conditions for positive and negative effects δ_i^+ *and* δ_i^- *from ordinary effect axioms are symmetric (via negation).*

- universal *if for every effect axiom the variables of sort* OBJECT *occurring in terms of sort* FLUENT *are universally quantified (assuming negation normal form).*

- temporally determined *if all precondition axioms are of the form*

$$Poss(A(\bar{x}), s_1, s_2) \equiv \bigvee_i \pi_{A_i}[s_1] \wedge \varphi_i, \qquad (*)$$

where $\pi_A[s_1]$ *does not mention* s_2*, and each* φ_i *is an equality atom equating the time variable* s_2 *to a function with arguments among* s_1 *and* $\bar{x}$*.*

- deterministic *if*
  - *it is ramification free,*
  - *it is temporally determined with the additional requirement that* $(*)$ *contains a single disjunct (i.e.* $i = 1$*), and*
  - *for every effect axiom*

$$Poss(A(\bar{x}), s_1, s_2) \supset \eta_1[s_1, s_2] \vee \ldots \vee \eta_k[s_1, s_2]$$

the condition parts ϕ_i *and* ϕ_j *in* η_i *and* η_j *are mutually exclusive for* $i \neq j$*.*

- anytime *if it is sequential and action applicability is not tied to a specific time-point; that is* $\mathsf{D}_{\mathsf{Poss}} \cup \mathsf{D}_{\mathsf{aux}}$ *entail*

$$(\exists s_2)(Poss(a, s_1, s_2) \wedge (\forall f)[Holds(f, s_1) \equiv Holds(f, s_1')]) \supset \\ (\exists s_2')Poss(a, s_1', s_2').$$

The notion of a progressing action domain is very natural: We preclude action effects that take place in the past. The definition of a sequential domain forestalls axiomatizing truly concurrent actions. This restriction wrt. expressivity is necessary because of our reading of unexpanded ALPs as temporally ordered sequences.

Next consider an action that leads to a transition from time point s_1 to time point s_2: In ramification-free domains the effects of the action that materialize at time point s_2 depend only on properties that hold at time point s_1. Ramifications in the unifying action calculus are modeled as effects materializing at time point s_2 only if some other properties hold at s_2. We need this notion of ramification-free domains for the definition of deterministic domains.

Let us illustrate our notion of a deterministic domain by the following example:

Example 3.13 (Deterministic Domains)**.** *Assume we axiomatize the precondition of the action MoveSomeBlock as*

$$(\forall)Poss(MoveSomeBlock(x, y), s_1, s_2) \equiv s_2 = Do(MoveSomeBlock(x, y), s_1)$$

assuming it is always possible to move some block from x to y, and the effects accordingly as follows:[5]

$$\begin{aligned}(\forall)&Poss(MoveSomeBlock(x, y), s_1, s_2) \supset \\ &[(\forall f)[(\exists block_1)f = On(block_1, x) \ \wedge x \neq y \wedge \\ &\qquad (\neg\exists block_2)Holds(On(block_2, block_1), s_1) \ \wedge \\ &\qquad (\neg\exists block_3)(Holds(On(block_3, y), s_1) \vee y = Table)] \ \vee \\ &\quad (Holds(f, s_1) \ \wedge \neg\Phi(s_1)) \\ &\qquad\qquad \equiv Holds(f, s_2)].\end{aligned}$$

This effect axiom is deterministic according to our definition, although arguably it might contradict some of our intuitions wrt. determinism: We know that there is some block we can move, yet it might be a different one in every model. However, such a weak notion of determinism for action domains will resurface in the proof of proposition 5.1. This notion of determinism is determinism wrt. a specific interpretation — in every interpretation the action's effects are uniquely determined.

[5] For readability we denote the lengthy condition for a negative effect just by $\Phi(s_1)$.

The notion of a temporally determined action domain is implicit in the macro-expansion of action logic programs. The following example illustrates the difficulties arising otherwise:

Example 3.14 (Temporally Determined Domains)**.** *A (not temporally determined) precondition axiom of the form*

$$(\forall)Poss(A, s_1, s_2) \equiv s_2 > s_1$$

does not admit useful substitutions on s_2: Let the underlying time structure be given by the natural numbers and let $s_1 = 0$. Then each of the infinitely many $n \in \mathbb{N}$ such that $n > 0$ constitutes a potential endpoint for the action's duration. However, we cannot enumerate these infinitely many potential endpoints with the help of a disjunctive substitution. In contrast, with a branching time structure as in the Situation or the Fluent Calculus, precondition axioms are always of the form

$$(\forall)Poss(A(\bar{x}), s_1, s_2) \equiv \pi(\bar{x}, s_1) \wedge s_2 = Do(A(\bar{x}), s_1),$$

where $\pi(\bar{x}, s_1)$ does not contain s_2; these domain axiomatizations are necessarily temporally determined. The precondition axiom

$$(\forall)Poss(A, s_1, s_2) \equiv s_2 = s_1 + 10 \vee s_2 = s_1 + 20$$

illustrates how actions with variable duration can be axiomatized.

The idea of anytime action domains, too, is implicit in the definition of macro-expansion of action logic programs. Let us likewise illustrate this point by means of an example:

Example 3.15 (Anytime Domains)**.** *Let the underlying time structure be the natural numbers. Suppose there are two actions A and B, with respective precondition axioms $Poss(A, s_1, s_2) \equiv s_1 = 0 \wedge s_2 = 1$ and $Poss(B, s_1, s_2) \equiv s_1 = 2 \wedge s_2 = 3$. Intuitively, it should be possible to do A and then B. The query* `?- do(a), do(b)` *is macro-expanded to $(\exists)Poss(A, 0, s_1') \wedge Poss(B, s_1', s_2')$, however, which does not follow. This kind of problem is precluded by anytime domains.*

We have seen that the macro-expansion of ALPs does not harmonize with domain axiomatizations that are not anytime or temporally determined. There are a number of options: We can adjust macro-expansion by only imposing a partial order on the time variables. If we stick with the defined macro-expansion, and at the same time do not restrict the underlying action theories then the desired planning completeness result is precluded. The third option is captured by the following definitions:

Definition 3.9 (Admissible Domain Axiomatization)**.** *A domain axiomatization* D *is admissible if it is modular, sequential, temporally determined, anytime, universal, symmetric, and ramification free. Furthermore, we restrict the axiomatization of the underlying time structure to be given by by the second order axiomatization of situations (cf. [Reiter, 2001a]).*

These conditions for admissible action domains are already quite restrictive. On the one hand, they preserve all of Situation and Fluent Calculus; on the other hand, they rule out the Event Calculus, which is based on a different time structure. They also rule out modeling effect axioms with occluded fluents — cf. the discussion of how occlusions can be modeled in the UAC at the end of definition 2.11. Finally, observe that temporally determined domains that are based on the time structure of situations do not allow to model actions with varying duration.

3.5.2 Strong Planning Completeness

But ssuming that an action domain under consideration D is admissible we can obtain the following result:

Theorem 3.3 (Strong Planning Completeness)**.** *Let* P *be the generic ALP encoding of a planning problem*

```
strategy :- ?(Phi).
strategy :- do(A), strategy.
```

familiar from example 3.1, and let the query ϱ *be* `?- strategy.`*. Assume that* $\mathsf{D} \models (\exists s)HOLDS(\phi, s)$ *and* $\mathsf{P} \cup \mathsf{D} \models (\forall)\sigma \supset \varrho$ *and* σ *is satisfiable wrt.* D*, where* ϕ *does not quantify over fluents. Then there exist successful derivations of the query* $Strategy(S_0, s)$ *in CLP(*D*) with computed answers* $\sigma_1, \ldots, \sigma_n$ *such that* $\mathsf{D} \models (\forall)(\sigma \supset (\sigma_1 \vee \ldots \vee \sigma_n))$*. Moreover, the plans computed by these derivations can be combined into a (disjunctive) plan achieving the planning goal* ϕ*. If in addition the domain axiomatization* D *is also query-complete a similar result for LP(*D*) is obtained.*

Before giving a proof of the theorem let us fix the notation for plans:

Definition 3.10 (Plan, Disjunctive Plan)**.** *A valid plan that is achieving the goal* $HOLDS(\phi, s)$ *in an action theory* D *is a conjunction* $\bigwedge_{i=1..k} Poss(a_i, s_{i_1}, s_{i_2})$ *of Poss atoms such that*

- *each* a_i *is a non-variable action term (possibly containing variables); and*
- *each* s_{i_j} *is a non-variable time term such that*
  - $s_{1_1} = S_0$*;*
  - $s_{i+1_1} = s_{i_2}$*; and*

– $\mathsf{D} \models \mathit{HOLDS}(\phi, s_{k_2})$.

A valid disjunctive plan is a disjunction of valid plans such that the action theory entails that the goal holds at one of the plan endpoints. The reading of conditional plans as trees of Poss atoms is readily unfolded to an equivalent reading as a disjunctive plan.

Sketch. The proof consists of two parts:

First, we have to show that if $\mathsf{D} \models (\exists s)\mathrm{HOLDS}(\phi, s)$ then there exists a disjunctive plan with possible final time-points $s_{i=1..k}$ such that

$$\mathsf{D} \models (\exists s)\mathrm{HOLDS}(\phi, s) \wedge \bigvee_i s = s_i.$$

That is, if there exists a timepoint satisfying the goal, then there also is a (disjunctive) plan achieving the goal.

In a second step we have to show that this disjunctive plan can be found by the generic planning ALP.

Concerning the first step, we note that admissible action theories admit a form of reasoning by regression that is very similar to the standard reasoning by regression in Reiter's Situation Calculus [Reiter, 2001a]. For this we need that admissible action theories are modular, sequential, temporally determined, anytime, universal, symmetric, and ramification free. Moreover, we exploit the condition that the planning goal does not quantify over fluents.

To see how this regression reasoning works assume given an atom of the form $\mathrm{Holds}(F(\vec{x}), \mathrm{Do}(A, s))$[6], and the respective precondition and effect axiom; the last of these will be of the form

$$\mathrm{Poss}(A(\vec{x}), s_1, s_2) \supset \bigvee_i (\exists \vec{y}_i)(\phi_i[s_1] \wedge (\forall f)[\delta_i[s_1] \equiv \mathrm{Holds}(f, s_2)].$$

Let the precondition axiom be given by

$$\mathrm{Poss}(A, s_1, s_2) \equiv \pi[s_1] \wedge s_2 = \mathrm{Do}(A, s_1).$$

Then for regression we instantiate the effect axiom by $f = F(\vec{x})$ and replace the atom $\mathrm{Holds}(F(\vec{x}), \mathrm{Do}(A, s))$ by the appropriately instantiated

$$\bigvee_i (\exists \vec{y}_i)(\phi_i[s] \wedge (\forall f)[\delta_i[s] \wedge \pi[s]]).$$

This idea is then extended to complex formulas in the spirit of [Reiter, 2001a].

[6]The situation term s shall denote any non-variable situation term.

Next we observe that it has already been proven for the Situation Calculus in [Savelli, 2006] that goal achievability implies the existence of a (disjunctive) plan. We can show that admissible action domains likewise admit reasoning by regression. Given this, and that the underlying time structure is given by a second order axiomatization of situations the respective proof for the Situation Calculus is readily adapted to our setting.

Given this first step of the proof, the second step is immediate: If there exists a (disjunctive) plan achieving the goal the generic planning ALP will find it.⊣ □

Of course, the generic planning ALP can find a (disjunctive) plan achieving a goal if it exists — even if the underlying action domain is not admissible. In particular the very same disjunctive plans will be found no matter whether the axiomatization of situations in an admissible action domain is second order or first order (replacing the second order induction axiom by a first order induction scheme). It is only the proof of the correspondence between goal achievability and plan existence from [Savelli, 2006] that does not go through unmodified in the first order case.

3.5.3 ALPs for Conditional Planning

Next follows a guide for writing ALPs that are meant to find conditional plans. In general, these will work only if a clause like

```
strategy :- do(A), strategy.
```

is included. Inferring a disjunctive substitution on the action variable `A` corresponds to conditional branching in the planning tree. However, we cannot write an ALP clause specifying which branching we would like to try. For example, we might want to try whether applying either action A_1 or action A_2 will achieve our goal. But as an ALP we cannot directly write something like

```
strategy :- (do(a1) \/ do(a2)), strategy.
```

If ϕ_i is the precondition of action A_i we can of course write

```
strategy :-
     ?((Phi1 /\ A = a1) \/ (Phi2 /\ A = a2), do(A),
     strategy.
```

For this, however, the programmer needs intimate knowledge of the action domain axiomatization — something that the ALP framework tries to make superfluous. We can also try using the Lloyd-Topor transformation for disjunction in logic programs. This results in a program

```
strategy :- do(a1), strategy.
strategy :- do(a2), strategy.
```

However, if we use this program in CLP(D) we need two derivations (via the constraint rule each) in order to derive that the strategy is successful. If we use an action variable `A` instead we obtain a single derivation using a most general disjunctive substitution.

Hence, as a rule of thumb, in an ALP for solving conditional planning problems a clause including a literal `do(A)` should be included. Making this clause the last in the program ensures that unconditional plans are preferred over conditional plans in the search for solutions.

3.5.4 Conditional versus Conformant Planning

As discussed in the preceding sections LP(D) can be used to solve unconditional planning problems, whereas CLP(D) covers the conditional case, where we want to find one suitable plan for every possible state of affairs. If we restrict state representation to propositional logic our notions of unconditional and conditional planning problems coincide with those studied by the planning research community. The planning research community has studied a third type of planning problems that is situated between the conditional and the unconditional case. It is this problem of conformant planning that we will address in this section.

As in conditional planning we admit non-query-complete background theories, and, hence, disjunctive or existential information. The problem of conformant planning now consists of finding a single plan that will achieve the planning goal no matter what the real state of affairs is. The underlying assumption is that the agent is devoid of sensors (something we will discuss next), and hence cannot determine the actual state of the world when executing the plan. ALPs also provide a logical characterization of conformant planning, again generalized to full first order state representation:

Proposition 3.3 (Conformant Planning Completeness)**.** *First recall the generic encoding of a planning problem as an ALP* P

```
strategy :- ?(Phi).
strategy :- do(A), strategy.
```

familiar from example 3.1, and then let the query ϱ be `?- strategy.`*. Further let* D *be a non-query-complete, but admissible domain (cf. definitions 3.3 and 3.9). Assume that there is a conformant (i.e. non-disjunctive) plan achieving HOLDS(ϕ, s). Then there is a successful derivation of the query* `?- strategy.` *in the LP(*D*) calculus.*

The main purpose of the ALP framework is the specification of domain-dependent heuristics: For this, if the background action domain is not query-complete, then we may want to use the CLP(D) proof calculus for querying some state properties.

However, in its unrestricted version the CLP(D) calculus computes conditional, not necessarily conformant plans. In such a setting a version of our proof calculi tailored to conformant planning can be obtained as follows: We use the CLP(D) proof calculus on non-query-complete domains; but we restrict the evaluation of all Poss-literals, and of all HOLDS-literals that share variables with Poss-literals to the substitution rule using non-disjunctive substitutions.

3.6 Planning in the Presence of Sensing Actions

So far we have only dealt with actions that have effects on the domain of discourse — in some sense it is always possible to imagine a physical change effected by executing the actions. Sensing actions are usually seen to be of a different kind: If an agent performs a sensing action it does not change the world, but rather it learns about the state of the world. In this section we show how sensing can be accommodated in action domains based on the UAC, and, hence, be made available for ALPs.

Sensing actions have already been addressed in each of the big three action calculi: In the Event Calculus sensing actions have been considered, e.g., in [Shanahan, 2002] and [Shanahan and Randell, 2004], with a special emphasis on visual perception and the bidirectional flow of information between cognition on the logical level and low-level sensor data. Sensing actions have been introduced in the Fluent Calculus in [Thielscher, 2000]; the work on sensing in the Situation Calculus is summarized in [Scherl and Levesque, 2003]. Both the Fluent and the Situation Calculus approach combine sensing with a formalized notion of knowledge.

With this plethora of proposals for reasoning about sensing actions we face a dilemma: On the one hand we would like to have some method for reasoning about sensing actions; on the other hand the ALP framework stands apart by being independent from a particular action calculus, and by being based on classical first order logic only.

If we want to stick to these characteristics we cannot adopt the Situation Calculus approach to reasoning about knowledge and sensing: This is based on a special knowledge fluent, and the use of many different possible initial situations instead of just one. Both ideas are absent from Event and Fluent Calculus, as well as from all existing planning languages.

The formalized notion of knowledge in the Fluent Calculus relies upon the quantification over states. For this state terms (collections of fluents) have to be present in the action calculus — the UAC does not feature such terms.

In the following we propose a simple approach to reasoning about sensing actions in the UAC that requires no additional machinery.

3.6.1 Sensing Actions in the UAC

This approach is based on the following considerations: It is obvious that in offline planning we cannot sense properties of the real world. But it is reasonable to assume that for each sensor we know beforehand the finitely many, discrete sensing results that it can return.

We also make a — more contentious — simplifying assumption wrt. sensing fluents that are affected by exogenous actions. If a fluent might be affected by an action not under the agent's control then this fluent must be exempt from the frame assumption — otherwise in general a contradiction may occur. Assume, e.g., we sense the location of a block, and someone removes the block without our noticing it: If the frame assumption is applied to this block a contradiction is readily obtained.

However, always excluding all fluents that might be affected by an exogenous action from the frame assumption has its drawbacks, too: The agent immediately forgets what it has just learned about such fluents. Combining the perception of freely fluctuating properties with a solution to the frame problem in a single logical framework in an intelligent fashion is an intricate problem left open as a challenge for future work.

The formalized notions of knowledge in the Fluent and the Situation Calculus allow to make a distinction between the formal world model, and the agent's knowledge thereof: It is possible to express settings where the action domain entails a state property but the agent cannot infer that this property holds.

But for many applications it seems reasonable to identify the knowledge of the agent with all the (formalized) knowledge about the world. Moreover, if accordingly we do not leave the agent in the dark wrt. some aspects of the world model, then a formalized notion of knowledge does not appear to be strictly necessary: We can identify the knowledge of the agent with the logical consequences of its world model. For sensing actions in the UAC it is this approach that we will take.

Instead of saying that, for action domain D and state property ϕ, we have $\mathsf{D} \models \mathrm{KNOWS}(\phi, s)$ we can also say $\mathsf{D} \models \mathrm{HOLDS}(\phi, s)$. For the notion of "knowing whether" instead of $\mathsf{D} \models \mathrm{KNOWS}(\phi, s) \vee \mathrm{KNOWS}(\neg\phi, s)$ we can query whether $\mathsf{D} \models \mathrm{HOLDS}(\phi, s)$ or whether $\mathsf{D} \models \mathrm{HOLDS}(\neg\phi, s)$.

Our identifying knowledge with logical consequence gives rise to the so-called paradoxon of logical omniscience: The agent knows everything that is logically entailed by its world model. This has been criticized as counter-intuitive in the context of both the Fluent and the Situation Calculus [Reiter, 2001b, Thielscher, 2005d]. We do not wish to enter a discussion whether this alleged paradoxon actually is a problem here. A simplistic remedy for this issue would be to resort to a sound, but not complete reasoning method. Recent work on a more principled approach to this problem can be found in, e.g., [Liu et al., 2004].

Sensor Axioms

With these considerations in mind we opt for the following definition:

Definition 3.11 (Sensor Axiom)**.** *We introduce a subsort* SenseFluent *of sort* Fluent*, and finitely many function symbols S of that sort. For each sensor fluent function S we introduce a sensor axiom of the form*

$$(\forall s, \vec{x}) Holds(S(\vec{x}), s) \equiv \bigvee_{\vec{t} \in T} \vec{x} = \vec{t} \wedge \phi(\vec{x}, s),$$

where T is a set of ground object vectors, and $\phi(\vec{x}, s)$ is a state formula with free variables among $\vec{x}$ and s without occurrences of sense fluents.

The set T of ground object vectors describes the finitely many values the sensor can return, whereas the state formula $\phi(\vec{x}, s)$ is meant to describe what the sensor data mean — the meaning of sensor data may depend on both time-dependent and time-independent predicates. Observe that a sensor axiom is just a special kind of domain constraint. This observation leads us to define action domains with sensing in the UAC as follows:

Definition 3.12 (Action Domain with Sensing)**.** *An action domain* D *in the UAC is an action domain with sensing if*

- *there is a finite number of function symbols into sort* SenseFluent*; and*
- *for each fluent symbol S of that sort a sensor axiom is included in* D_{dc}*(we denote the set of all sensor axioms contained in* D_{dc} *by* $\mathsf{D}_{\mathsf{Sense}}$*).*

In our approach the purpose of the fluents of sort SenseFluent is to serve as an interface to the sensor; hence the restriction that there is one sensor axiom for each sense-fluent, and that no other sense fluents occur in that axiom.

It is worth pointing out that in the Situation Calculus we can only sense the truth value of sense fluents F. In the UAC sensing truth values of fluents can be modeled as follows:

$$\begin{aligned}(\forall x, \vec{y}, s)&\mathrm{Holds}(SF(x), s) \equiv \\ &[x = \mathrm{true} \wedge \mathrm{Holds}(F(\vec{y}), s)] \vee [x = \mathrm{false} \wedge \neg \mathrm{Holds}(F(\vec{y}), s)]\end{aligned}$$

The sensor axioms we just introduced are strictly more general, though. They are inspired by the sensing model of the Fluent Calculus.

3.6.2 Discussion of the Approach

In [Levesque, 1996] a number of properties that an approach to planning in the presence of sensing should exhibit have been discussed informally. In this section we evaluate our approach wrt. these criteria.

First, in [Levesque, 1996] three different reasons for the necessity of sensing actions are given: There may be incomplete information about the initial situation, there may be exogenous actions, and there may be uncertain effects of actions. All three settings allow a natural modeling in the UAC.

Redundant Branches in Plans Next it is argued that — in the presence of sensing — plans must at least be viewed as trees of actions, rather than as sequences. The same view is taken in ALPs. For this tree view of plans in general there is the problem that branches may be redundant. In the case of an unsuccessful redundant branch in a plan we might fail to recognize a valid plan, as is illustrated by the following example: Assume we have a number of sensing results t_i, and a number of actions a_i. Furthermore, assume that the goal can be achieved by action a_i if t_i is observed for all i except one — say t_1, a_1. Now, if the domain entails that t_1 is the only sensing result that cannot plausibly be observed then the plan "if t_i then do a_i" should be considered valid — the branch corresponding to t_1 is what we call a redundant branch. ALPs avoid this problem of redundant branches by appealing to the notion of a most general disjunctive substitution. For example, the query `?- ?(sense(X)), do(A).` in our running example will infer a conditional plan without the redundant branch a_1.

Epistemically Adequate Plans An important issue that also stems from [Levesque, 1996] is the notion of an epistemically adequate plan in the presence of sensing: A plan is *epistemically adequate* if an agent following that plan at run-time will always know precisely what to do. For illustration, assume we have two actions at our disposal: a_1 and a_2. Further assume that we will achieve our goal if we either execute a_1 or a_2 (but we do not know which one). Finally assume there is a sensing action that will identify which of the two actions is applicable. Now there is a big difference for an agent between a plan that says "Do a_1 or a_2", and a plan that says "Perform the sensing action, and execute the appropriate action". It is this latter plan that meets the criterion of being epistemically adequate.

Using the plain ALP framework by itself we cannot distinguish between these two types of plans. In either of the two cases we obtain a successful derivation. But this is easy to remedy: First, we generalize the notion of a plan to include the Holds atoms for sense fluents in addition to the Poss atoms. Next we introduce the following generic ALP, that captures epistemically adequate planning in the presence of sensing:

```
strategy :- ?(goal).
strategy :- ?(SF), strategy.
strategy :- do(A), strategy.
```

Put in words, we are done if we have achieved the goal — this is as before. Otherwise we

- obtain the (disjunction of) sensing results for sense fluent `SF`; or
- non-deterministically pick an action `A`

and continue planning. If furthermore we restrict the proof calculus so that disjunctive substitutions are only applicable for sense fluents, but not for actions, then we have obtained a characterization of epistemically adequate planning in the presence of sensing in the ALP framework.

Conditional vs. Cyclic Planning On a last important issue that has also already been discussed in [Levesque, 1996] our approach shows limitations: We can only infer conditional, tree-like plans of the form "if then". However, there are action domains where plans require "while" loops. One example of this is an agent that tries to chop down a tree, and after each blow has to perform sensing to see whether it has already been successful [Sardiña et al., 2004]. What a solution to this problem would have to look like in general is also discussed in [Sardiña et al., 2004] — where it is emphasized that currently no feasible approach to this issue is known.

Observe that this last issue is closely related to the notion of strong planning completeness (cf. section 3.5.2). In particular, in admissible action domains conditional plans without "while" are always enough. However, the wood chopping example has been axiomatized in Situation Calculus in [Sardiña et al., 2004]. This does not contradict the result on strong planning completeness because this axiomatization requires a second order initial state axiom — this observation has already been made in the discussion of this very issue in [Savelli, 2006].

3.6.3 ALPs for Planning in the Presence of Sensing Actions

Due to our particular approach to sensing actions for ALPs there are no major formal differences between domains with, and domains without sensing actions. Let us illustrate our approach to planning in the presence of sensors by the following example:

Example 3.16 (Colored Blocks World with Sensing)**.** *Consider a variant of the blocks world, where the blocks have colors. There are only two blocks, $Block_1$ and $Block_2$. Further assume that the agent is equipped with a sensor that allows it to*

observe the color of the blocks; that is the fluent $Coloring(block, x)$ *is of the sort* SenseFluent. *The following is the respective sensor axiom:*

$$\begin{aligned}(\forall block, color, s)Holds(Coloring(block, color), s) \equiv \\ (block = Block_1 \vee block = Block_2) \wedge \\ (color = Red \vee color = Blue \vee color = Green)\end{aligned}$$

Observe that here the sensing result does not depend upon the state of some fluent. The following domain constraint enforces that every block has exactly one color:

$$(\forall s)(\exists! color)Holds(Coloring(block, color), s)$$

The sensor axiom together with the domain constraint entail, e.g., that $Block_2$ *is green, if it is neither red nor blue. Let the initial state specification include*

$$\begin{aligned}&Holds(On(Block_1, Table), S_0) \wedge Holds(On(Block_2, Table), S_0) \wedge \\ &Holds(Free(Block_1), S_0) \wedge Holds(Free(Block_2), S_0) \wedge \\ &(Holds(Coloring(Block_1, Green), S_0) \vee Holds(Coloring(Block_2, Green), S_0))\end{aligned}$$

so that we only know that at least one of $Block_1$ *or* $Block_2$ *is green; note that we use the macro* $Holds(Free(Block_1), s) \stackrel{def}{=} \neg\exists Block_2 Holds(On(Block_2, Block_1), s)$. *Let the precondition and the effects of moving a block again be axiomatized as in example 2.1:*

$$\begin{aligned}&(\forall)Poss(Move(block_1, x, y), s_1, s_2) \equiv \\ &\quad Holds(On(block_1, x), s_1) \wedge x \neq y \wedge \\ &\quad Holds(Free(block_1), s_1) \wedge \\ &\quad (Holds(Free(y), s_1) \vee y = Table) \wedge \\ &\quad s_2 = Do(Move(block_1, x, y), s_1) \\ &(\forall)Poss(Move(block, x, y), s_1, s_2) \supset \\ &\quad [(\forall f)(f = On(block, y) \vee (Holds(f, s_1) \wedge f \neq On(block, x))) \equiv Holds(f, s_2)]\end{aligned}$$

The following ALP illustrates epistemically adequate offline planning with sensing actions — the goal is to build a tower of height two, with a green block on top:

```
strategy :- ?(on(X,Y) and on(Y,table) and coloring(X,green)).
strategy :- ?(coloring(X,green) and on(X,table) and free(X)),
            ?(on(Y,table) and X \= Y and free(Y)), do(move(X,Y)),
            strategy.
```

Put in words, unless we have reached the goal, we either

- *identify a green block on the table; or*
- *stack a known green block atop another block.*

Given the query `?- strategy.` *initially the last rule is not applicable because it is restricted to non-disjunctive substitutions for epistemically adequate planning. The first rule is not applicable either. After sensing we obtain two successful subderivations informing us that we can reach our goal no matter by either moving* $Block_1$ *atop of* $Block_2$ *or vice versa. By making the sense fluent in the sensing rule a variable it is even possible to plan the appropriate sensing actions that ensure that the goal can be achieved.*

In the above example the sense fluent Coloring is not affected by the effect axiom — it is a static property of the domain. As another example scenario where we sense dynamic properties of the domain consider an agent that is equipped with a sensor to identify different blocks: If the agent initially knows that one of two blocks is located on the table it can infer a disjunctive plan leading to a situation where it has a block in its gripper. Here the agent can both sense and change the location of blocks.

3.7 Offline vs. Online Execution of Action Logic Programs

In the context of sensing actions let us make some remarks on the issue of offline vs. online execution of ALPs. Basically, action logic programs can be used to solve two complementary tasks: They can be used to control an agent online, where the agent actually executes the actions that are part of a derivation; or the agent may use them offline, to infer a plan that helps it achieve its goals.

If we intend to use ALPs for the online control of an agent we can do this by non-deterministically picking one path in the proof tree of an action logic program. Essentially, this approach to online agent control is the same as the one taken in Golog and Flux.

In the case of the CLP(D) calculus the derivations are restricted to non-disjunctive substitutions, and the constraint rule may not be used at all. These restrictions are necessary because otherwise in general the agent would not know which action it is executing, or whether the action is even executable. The agent should behave more cautiously. Summing up, in the case of online execution, the LP(D) and the CLP(D) proof calculus really are identical. The resulting proof calculus is sound, but in the case of action domains that are not query-complete completeness of course does not hold.

But there is something more to the online execution of ALPs in the presence of sensing: The sensor axioms enumerate all the possible sensing results, and if we evaluate a sense fluent against the sensor axiom we obtain a disjunctive substitution.

In the online execution of an ALP, however, the idea is that we evaluate the sense fluent against the "real world", not against our axiomatization. The real world determines a single sensing result that applies.

In the CLP(D) proof calculus this corresponds to considering only a single case of the disjunctive substitution to continue the derivation with. Unfortunately, if such a derivation is successful it is not even sound wrt. the domain axiomatization D and the program P — all that we can say is that it is consistent with $\mathsf{D} \cup \mathsf{P}$. On the other hand, such a derivation is sound wrt. $\mathsf{D} \cup \mathsf{P}$ augmented by the real world; or, more precisely, $\mathsf{D} \cup \mathsf{P}$ augmented by the observed sensing results.

This is a problem shared by all approaches to the online control of agents equipped with sensors: Beforehand it is impossible to say which sensing results will be observed. But adding an actual sensing result to our formal world model D we obtain a new world model D'. Because our action theories are based on first order logic, which is monotonic, in D' we do not lose any consequences from D — but possibly we get some new consequences. This is the reason why a derivation in CLP(D) for domains with sensing is only sound wrt. the program and the action domain augmented by the sensing results.

Now assume that an action domain contains disjunctive or merely existential information wrt. some properties of the world: An ALP that is meant for online agent control may try to ensure with the help of sensing actions that the actual values of these properties are known at run-time. Observe that for this it is of paramount importance that the values of fluents mentioned in a sensor axiom never contradict the agent's world model.

Let us conclude this section with the remark that the combination of online execution and the computation rule of negation as finite failure is not unproblematic: We read negated goals as a goal not being achievable by the execution of actions. In the online setting this does not seem to make much sense. Hence, for this setting, negation as finite failure can only safely be used for program atoms that do not depend on the time-dependent special atoms.

4 ALPprolog— LP(D) over Deterministic, Propositional Fluent Calculus Domains with Sensing

In this chapter we present ALPprolog[1] — an implementation of the ALP framework atop of action theories in a version of the Fluent Calculus that

- uses (a notational variant of) propositional logic for describing state properties;
- is restricted to actions with ground deterministic effects; and
- includes sensing actions.

The intended application domain for ALPprolog is the online control of agents in dynamic domains with incomplete information — hence all the remarks on online reasoning from section 3.7 apply.

ALPprolog is inspired by, and closely related to, dialects of the two prominent action programming languages Flux [Thielscher, 2005a] (cf. also section 6.2) and Golog [Levesque et al., 1997] (cf. also section 6.1).

In a sense it covers the middle-ground between special and full Flux. In Special Flux the expressivity is limited to conjunctions of ground, positive atoms: negated atoms are expressed using the closed world assumption. ALPprolog also uses a ground state representation, but on the other hand also admits disjunctive state knowledge, negation, sensing, and incomplete knowledge (open world semantics). It does not reach the expressivity of full Flux in that it does not support arbitrary, possibly non-ground terms, quantifiers, and an explicit notion of knowledge. It transcends the expressivity of Flux in that it fully supports disjunction — Flux (depending on the dialect) allows only one negative literal in disjunctions (or none at all).

From the variant of Golog that supports open-world planning [Reiter, 2001a] ALPprolog takes the representation of state knowledge via prime implicates. Here, the major difference is that ALPprolog uses progression whereas Golog uses regression.

The fragment of the Fluent Calculus that ALPprolog supports is tailored for an efficient implementation using Prolog list operations — the actual implementation is

[1] The name refers to both the implementation language Prolog, and the underlying logic (*Pro*positional *Log*ic).

based on ECLiPSe Prolog [ECLiPSe Implementors Group, 2009]. One major design objective for ALPprolog was to both extend the expressivity of Special Flux and to retain (some of) its practical efficiency.

4.1 ALPprolog Programs

An ALPprolog program is an ALP that respects the following restrictions on the `?(Phi)` atoms in the program:

- All occurrences of non-fluent expressions in ϕ are positive.
- So called sense fluents $S(\vec{x})$ that represent the interface to a sensor may only occur in the form `?(s(X))`. Sense fluents are formally introduced below.

The following will be our running example of a ALPprolog program throughout this section:

Example 4.1. Consider an agent whose task is to find gold in a maze. For the sake of simplicity, the states of the environment shall be described by a single *fluent* (i.e., state property): $At(u, x)$ to denote that $u \in \{Agent, Gold\}$ is at location x. The agent can perform the action $Go(y)$ of going to location y, which is possible if y is adjacent to, and accessible from, the current location of the agent. The fluent and action are used as basic elements in the following agent logic program. It describes a simple search strategy based on two parameters: a given list of locations (choice points) that the agent may visit, and an ordered collection of backtracking points.

```
explore(Choicepoints,Backtrack) :-          % finished, if
   ?(at(agent,X)), ?(at(gold,X)).           % gold is found

explore(Choicepoints,Backtrack) :-
   ?(at(agent,X)),
   select(Y,Choicepoints,NewChoicepoints),  % choose a direction
   do(go(Y)),                               % go in this direction
   explore(NewChoicepoints,[X|Backtrack]).  % store the choice made

explore(Choicepoints,[X|Backtrack]) :-      % go back one step
   do(go(X)),
   explore(Choicepoints,Backtrack).

select(X,[X|Xs],Xs).
select(X,[Y|Xs],[Y|Ys]) :- select(X,Xs,Ys).
```

Suppose we are given a list of choice points `C`, then the query `:- explore(C,[])` lets the agent systematically search for gold from its current location: the first

clause describes the base case where the agent is successful; the second clause lets the agent select a new location from the list of choice points and go to this location (the declarative semantics and proof theory for $\texttt{do}(\alpha)$ will require that the action is possible at the time of execution); and the third clause sends the agent back using the latest backtracking point.

Because ALPprolog programs are meant for online execution the programmer must ensure that no backtracking over action executions occurs, by inserting cuts after all action occurrences. Observe that this applies to sensing actions, too. It is readily checked that — after the insertion of cuts — the ALP from example 4.1 satisfies all of the above conditions.

4.2 Propositional Fluent Calculus

In this section we introduce the announced propositional fragment of the Fluent Calculus. The discussion of sensing is deferred until section 4.3.

For ease of modeling we admit finitely many ground terms for fluents and objects, instead of working directly with propositional letters. An action domain D is then made propositional by including the respective domain closure axioms. For actions, objects, and fluents unique name axioms are included — hence we can avoid equality reasoning.

The basic building block of both the propositional Fluent Calculus and ALPprolog are the so-called prime implicates of a state formula $\phi(s)$:

Definition 4.1 (Prime Implicate)**.** *A clause ψ is a prime implicate of ϕ if and only if*

- *it is entailed by ϕ;*
- *it is not a tautology; and*
- *it is not entailed by another prime implicate.*

The prime implicates of a formula are free from redundancy — all tautologies and implied clauses have been deleted. For any state formula an equivalent *prime state formula* can be obtained by first transforming the state formula into a set of clauses, and by then closing this set under resolution, and the deletion of subsumed clauses and tautologies.

Prime state formulas have the following nice property: Let ϕ be a prime state formula, and let ψ be some clause (not mentioning auxiliary predicates); then ψ is entailed by ϕ if and only if it is subsumed by some prime implicate in ϕ, a fact that has already been exploited for Golog [Reiter, 2001b, **?**]. This property will allow

us to reduce reasoning about state knowledge in ALPprolog to simple list look-up operations.

Formally the propositional version of the Fluent Calculus is defined as follows.

Definition 4.2 (Propositional Fluent Calculus Domain)**.** *We stipulate that the following properties hold in propositional Fluent Calculus domains:*

- *The initial state* $\mathsf{D}_{\mathsf{Init}}$ *is specified by a ground prime state formula.*
- *The state formulas* $\phi(s_1)$ *in action preconditions* $Poss(a, s_1, s_2) \equiv \phi(s_1) \wedge s_2 = Do(a, s_1)$ *are prime state formulas.*
- *The effect axioms are of the form*

$$Poss(A(\vec{x}), s_1, s_2) \supset \bigvee_k (\Phi_k[s_1] \wedge (\forall f)[(\bigvee_i f = f_{ki} \vee (Holds(f, s_1) \wedge \bigwedge_j f \neq g_{kj})) \equiv Holds(f, s_2)]),$$

  where each $\Phi_k[s_1]$ *is a prime state formula. This implies that existentially quantified variables that may occur in case selection formulas (cf. definition 2.2) have been eliminated by introducing additional cases.*

- *Only so-called* modular *domain constraints [Herzig and Varzinczak, 2007] may be included. Very roughly, domain constraints are modular if they can be compiled into the agent's initial state knowledge, and the effect axioms ensure that updated states also respect the domain constraints. In the Fluent Calculus this holds if the following two conditions are met: The first condition, (4.1), essentially says that for every state at some time* S *which is consistent with the domain constraints and in which an action* $A(\vec{x})$ *is applicable, the condition* $\Phi_i[S]$ *for at least one case* i *in the effect axiom for* A *holds. Condition (4.2) requires that any possible update leads to a state that satisfies the domain constraints. Formally, let* S, T *be constants of sort* Time*. For a domain axiomatization with domain constraints* D_{dc}*, precondition axioms* $\mathsf{D}_{\mathsf{Poss}}$*, and effect axioms* $\mathsf{D}_{\mathsf{Effects}}$ *the following must hold for every action* $A(\vec{x})$ *with effect axiom (2.2): There exists* $i = 1, \ldots, k$ *such that*

$$\models \mathsf{D}_{\mathsf{dc}}[S] \wedge \pi_A[S] \wedge (\exists \vec{y}_i)\Phi_i[S], \tag{4.1}$$

  and for every such i*,*

$$\models \mathsf{D}_{\mathsf{dc}}[S] \wedge \pi_A[S] \wedge \eta_i[S, T] \supset \mathsf{D}_{\mathsf{dc}}[T] \tag{4.2}$$

  An in-depth treatment of modular domain constraints in the UAC can be found in [?]. The problem with general domain constraints is that they greatly complicate reasoning.

- *Auxiliary time-independent axioms may be included if they can faithfully be represented in the Prolog dialect underlying the implementation. This deliberately sloppy condition is intended to allow the programmer to use her favorite Prolog library. However, we stipulate that auxiliary predicates occur only positively outside of* $\mathsf{D}_{\mathsf{aux}}$ *in the action domain* D*in order to ensure that they can safely be evaluated by Prolog. They also must not occur in the initial state formula at all. The update mechanism underlying ALPprolog can handle only ground effects. Hence, if auxiliary atoms are used in action preconditions, case selection formulas of effect axioms, then it is the burden of the programmer to ensure that these predicates always evaluate to ground terms on those variables that also occur in the action's effects.*

On the one hand clearly every propositional Fluent Calculus domain can be transformed to this form. On the other hand it is well known that in general compiling away the quantifiers in a state formula can result in an exponential blow-up, as can the conversion to conjunctive normal form. We believe that the simplicity of reasoning with prime implicates outweighs this drawback.

Propositional action domains can still be non-deterministic. For example, for an applicable action two different cases may be applicable at the same time. The resulting state would then be determined only by the disjunction of the cases' effects. What is more, it would be logically unsound to consider only the effects of one of the cases. For the online control of agents in ALPprolog we stipulate that for an applicable action at most a single case applies, greatly simplifying the update of the agent's state knowledge.

Definition 4.3 (Deterministic Propositional Fluent Calculus)**.** *A propositional Fluent Calculus domain is deterministic if the following holds: Let a be an applicable ground action. Then there is at most one case of the action that is applicable in the given state.*

For example, an action theory is deterministic if for each effect axiom all the cases are mutually exclusive. Next assume we have an applicable deterministic action with e.g. two case selection formulas $\phi(s)$ and $\neg\phi(s)$, where neither case is implied by the current state. Here, instead of updating the current state with the disjunction of the respective effects, ALPprolog will employ incomplete reasoning.

4.3 Propositional Fluent Calculus with Sensing

We make the following assumptions concerning sensing: At any one time, a sensor may only return a single value from a fixed set $\mathcal{R}$ of ground terms, the *sensing results*. However, the meaning of such a sensing result may depend upon the concrete situation of the agent.

Example 4.1. (continued) *Assume that now one of the cells in the maze contains a deadly threat to our gold-hunting agent. If the agent is next to a cell containing the threat she perceives a certain smell, otherwise she doesn't: The sensing result indicates that one of the neighboring cells is unsafe; the actual neighboring cells, however, are only determined by the agent's current location.*

We adopt the following formulation of sensor axioms:

Definition 4.4 (Sensor Axiom)**.** *A sensor axiom in a propositional Fluent Calculus domain is of the form*

$$(\forall s, x, \vec{y})\mathit{Holds}(S(x), s) \equiv \bigvee_{R \in \mathcal{R}} x = R \wedge \phi(x, \vec{y}, s) \wedge \psi(x, \vec{y}, s),$$

for a ground set of sensing results $\mathcal{R}$. Here $\phi(x, \vec{y}, s)$ is a prime state formula that selects a meaning of the sensing result R, whereas the pure prime state formula $\psi(x, \vec{y}, s)$ describes the selected meaning. We stipulate that sensor axioms (which are a form of domain constraint) may only be included if they are modular.

Clearly $\phi(x, \vec{y}, s)$ should be chosen such as to denote a property that is uniquely determined in each state. If auxiliary axioms are used in $\phi(x, \vec{y}, s)$ it is again the burden of the programmer to ensure that these evaluate to ground terms in order to ensure that a ground state representation can be maintained.

Example 4.1. (continued) *The following is the sensor axiom for our gold-hunter:*

$$\begin{aligned}(\forall)\mathit{Holds}(\mathrm{PerceiveSmell}(x), s) \equiv\\ x = \mathrm{true} \ \wedge \mathit{Holds}(\mathrm{At}(\mathrm{Agent}, y), s) \wedge \mathrm{Neighbors}(y, \vec{z}) \wedge \bigvee_{z \in \vec{z}} \mathit{Holds}(\mathrm{ThreatAt}(z), s)\\ \vee\\ x = \mathrm{false} \ \wedge \mathit{Holds}(\mathrm{At}(\mathrm{Agent}, y), s) \wedge \mathrm{Neighbors}(y, \vec{z}) \wedge \bigwedge_{z \in \vec{z}} \neg\mathit{Holds}(\mathrm{ThreatAt}(z), s))\end{aligned}$$

Theoretically, the combination of sensing with the online control of an agent is quite challenging: For offline reasoning it is enough to consider the disjunction of all possible sensing results, and this is logically sound. In the online setting, however, upon the observation of a sensing result we henceforth have to accept this result as being true; that is, we at runtime *add* the result to the action theory, something which is logically unsound. On the other hand, it also does not make sense to stipulate that the sensing result be known beforehand.

4.4 Action Theory Representation

We continue by describing how the underlying action theory is represented in ALPprolog. As basic building block we need a representation for prime state formulas. For notational convenience we will represent $(\neg)\mathrm{Holds}(f, s)$ literals by the (possibly negated) fluent terms only, and, by an abuse of terminology, we will call such a term $(\neg)f$ a fluent literal. A convenient Prolog representation for such a state formula is a list, where

- each element is either a literal (i.e. a unit clause), or
- a list of at least two literals (a non-unit clause).

In the following we call such a list a PI-list.

Definition 4.5 (Action Theory Representation)**.** *Action theories as defined in definition* **??** *are represented in ALPprolog as follows:*

- *The initial state is specified by a Prolog fact* `initial_state(PI-List).`*, where* `PI-List` *mentions only ground fluent literals. Domain constraints other than sensor axioms have to be compiled into* `PI-List`*.*
- *For each action a, a Prolog fact* `action(A,Precond,EffAx).` *has to be included, where*
  - `A` *is an action function symbol, possibly with object terms as arguments;*
  - `Precond` *is a PI-list, the action's precondition;*
  - `EffAx` *is a list of cases for the action's effects with each case being a pair* `Cond-Eff`*, where the effect's condition* `Cond` *is a PI-list, and the effects* `Eff` *are a list of fluent literals; and*
  - *all variables in* `EffAx` *also occur in* `Precond`*.*
- *If present, auxiliary axioms* $\mathsf{D}_{\mathsf{aux}}$ *are represented by a set of Prolog clauses. The predicates defined in the auxiliary axioms must be declared explicitly by a fact* `aux(Aux).`*, where* `Aux` *denotes the listing of the respective predicate symbols.*

The sensor axioms are represented as Prolog facts `sensor_axiom(s(X),Vals)`*, where*

- `s` *is a sense fluent with object argument* `X`*; and*
- `Vals` *is a list of* `Val-Index-Meaning` *triples, where*
  - `Val` *is a pair* `X-result_i`*, where* `result_i` *is the observed sensing result;*
  - `Index` *is a PI-list consisting of unit clauses; and*

– `Meaning` *is a PI-list, mentioning only fluent literals and only variables from* `Val` *and* `Index`*.*

The sense fluents have to be declared explicitly by a fact `sensors(Sensors).`*, where* `Sensors` *is a listing of the respective function symbols. This is necessary in order to distinguish sense fluents, ordinary fluents, and auxiliary predicates in PI-lists.*

4.5 Reasoning for ALPprolog

Reasoning in ALPprolog works as follows: For evaluating the program atoms we readily resort to Prolog. The reasoner for the action theory is based on the principle of progression. Setting out from the initial state, upon each successful evaluation of an action's precondition against the current state description, we update the current state description by the action's effects.

Reasoning about the action comes in the following forms:

- Given a ground applicable action a, from the current state description $\phi(s_1)$ and the action's positive and negative effects compute the description of the next state $\psi(s_2)$ (the update problem).
- Given a description $\phi(s)$ of the current state, check whether $\{\phi(s)\} \cup \mathsf{D}_{\mathsf{aux}} \models \psi(s)$, where $\psi(s)$ is some state formula in s, but not a sense fluent (the entailment problem).
- For a sensing action, i.e. a query $\mathrm{Holds}(S(x), s)$, integrate the sensing results observed into the agent's state knowledge (the sensing problem).

In the following we consider each of these reasoning problems in turn.

4.5.1 The Update Problem

It turns out that solving the update problem is very simple. Let `State` be a ground PI-List, and let `Update` be a list of ground fluents. The representation of the next state is then computed in two steps:

(1) First, all prime implicates in `State` that contain either an effect from `Update`, or its negation, are deleted, resulting in `State1`.

(2) The next state `NextState` is given by the union of `State1` and `Update`.

Starting from a ground initial state only ground states are computed.

The correctness of this procedure can be seen e.g. as follows: In [Liu et al., 2006, Drescher et al., 2009] algorithms for computing updates in a Fluent Calculus based upon Description Logics have been developed. The above update algorithm constitutes a special case of these algorithms.

4.5.2 The Entailment Problem

When evaluating a clause ψ against a ground prime state formula ϕ, ψ is first split into the fluent part ψ_1, and the non-fluent part ψ_2. It then holds that ψ is entailed by ϕ if there is a ground substitution θ such that

- $\psi_1\theta$ is subsumed by some prime implicate in ϕ; or
- some auxiliary atom $P(\vec{x})\theta$ from ψ_2 can be derived from its defining Prolog clauses.

Computing that the clause ψ_1 is subsumed by ϕ can be done as follows:

- If ψ_1 is a singleton, then it must be a prime implicate of ϕ (modulo unification).
- Otherwise there must be a prime implicate in ϕ that contains ψ_1 (modulo unification).

Hence the entailment problem for ALPprolog can be solved by `member`, `memberchk`, and `subset` operations on sorted, duplicate-free lists.

The following example illustrates how reasoning in ALPprolog can be reduced to simple operation on lists. It also illustrates the limited form of reasoning about disjunctive information available in ALPprolog:

Example 4.2 (Disjunctions and Substitutions in ALPprolog). Assume that the current state is given by `[[at(gold,4),at(gold,5)]]`. Then the query `?([at(gold,X)])` fails, because we don't consider disjunctive substitutions. However, on the same current state the query `?([[at(gold,X),at(gold,Y)]])` succeeds with `X=4` and `Y=5`.

4.5.3 The Sensing Problem

Sensing results have to be included into the agent's state knowledge every time a sensing action is performed, i.e. a literal `?(s(X))` is evaluated. This works as follows:

- First we identify the appropriate sensor axiom `sensor_axiom(s(X),Vals)`.
- Next we identify all the `[X-result_i]-Index-Meaning` triples in `Vals` such that `result_i` matches the observed sensing result, and unify `X` with `result_i`.

- After that we find the unique `Index-Meaning` such that the current state entails `Index`.
- Finally, we adjoin `Meaning` to the current state and transform this union to a PI-list.

4.6 Soundness of ALPprolog

At the end of section 4.3 we have already mentioned that adding sensing results to the action theory at runtime makes the subsequent reasoning logically unsound wrt. the original program plus action theory. If we add the set of sensing results observed throughout a run of an ALPprolog program, however, then we can obtain the following soundness result:

Proposition 4.1 (Soundness of ALPprolog)**.** *Let* P *be a ALPprolog program on top of an action domain* D*. Let* Σ *be the union of the sensor results observed during a successful derivation of the ALPprolog query* ϱ *with computed answer substitution* θ*. Then* $\mathsf{D} \cup \mathsf{P} \cup \Sigma \models \varrho\theta$*.*

Proof. **(Sketch)** SLD-resolution is sound for ordinary program atoms. $(\exists)\text{Holds}(\phi)$, where ϕ is not a sense fluent, is only derived if there is a substitution θ such that $\mathsf{D} \models (\forall)\text{Holds}(\phi)\theta$. For sense fluents we need the observed sensing result as an additional assumption. □

4.7 Evaluation

Let us now turn to the evaluation of ALPprolog. First we compare it to Special Flux, on a very simple action domain. We then evaluate ALPprolog's performance against that of Golog and Flux on the well-known Wumpus world. Finally, we take a glimpse at promising future application domains for ALPprolog.

4.7.1 Example I — The Mailbot Domain

The protagonist of the mailbot domain [Thielscher, 2005a] is a robot that carries packages. It lives in a very simple world: All the rooms are linearly arranged into an aisle. The robot can perform three actions: go up/down the aisle, pickup a package, or deliver it. The domain is very simple — it can be represented by a conjunction of ground literals. We use this example for two purposes: First it serves to illustrate how a real action domain is represented in ALPprolog, with the exception of sensor axioms — these will be illustrated in the next example. Second, we use it to compare the performance of ALPprolog and Special Flux.

Example 4.3 (Mailbot Domain in ALPprolog). We can specify the simple mailbot domain in ALPprolog as follows:

```
initial_state([at(room1), empty(slot1), request(room1,room2),
               connected(room1,room2), connected(room2,room1)]).

action(deliver(Slot),
       [ [at(Room), carries(Slot,Room)]-
         [not(carries(Slot,Room)), empty(Slot)] ]).

action(pickup(Slot, Room),
       [ [empty(Slot), at(Room2), request(Room2,Room)]-
         [not(request(Room2,Room)), not(empty(Slot)),
          carries(Slot1,Room)] ]).

action(go(Room),
       [ [at(Room2), connected(Room2,Room)]-
         [not(at(Room2)), at(Room)] ]).
```

We omit the simple ALP that makes the robot fulfill all requests.

Our experiments comparing the performance of ALPprolog to that of Special Flux have led to the following conclusions: The run-time of the former is a small multiple of that of the latter if we explicitly include all the negative information — something which strictly speaking is is not necessary, though. Most importantly, ALPprolog has proven its scalability.

The mailbot domain has also been used in [Thielscher, 2004] to evaluate the performance of Special Flux wrt. that of Golog. Because ALPprolog behaves quite similar to Special Flux, the conclusions drawn there also apply to ALPprolog: The more actions have to be executed to reach the goal, the more the progression approach of ALPprolog gains over the regression approach of Golog.

4.7.2 Example II — The Wumpus World

The Wumpus World [Russell and Norvig, 2003] is a well-known challenge problem in the reasoning about action community. It consists of a grid-like world: cells may contain pits, one cell contains gold, and one the fearsome Wumpus. The agent dies if she enters a cell containing a pit or the Wumpus. But she carries one arrow so that she can shoot the Wumpus from an adjacent cell. If the agent is next to a cell containing a pit (the Wumpus), she can detect that one of the surrounding cells contains a pit (the Wumpus), but doesn't know which one: She perceives a breeze if she is in a cell next to a pit, and she perceives a stench if she is in a cell next to the Wumpus. She knows the contents of the already visited cells — in particular

she can detect the gold if she is in the same cell. Getting the gold without getting killed is the agent's goal. See figure 4.1[2] for an example scenario.

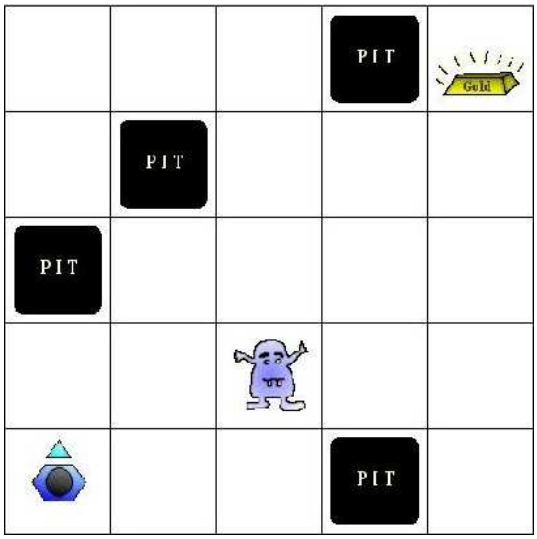


Figure 4.1: A Wumpus World

Example 4.4 (Sensor Axioms in ALPprolog). Using the Wumpus world we illustrate how sensor axioms are modeled in ALPprolog— the possible outcomes of sensing whether an adjacent cell contains a pit are represented as follows :

```
sensor_axiom(breeze(X),
          [ [X-true] -
            [at(agent,Y),neighbors(Y,[C1,C2,C3,C4])] -
            [[pit(C1),pit(C2),pit(C3),pit(C4)]],

            [X-false] -
            [at(agent,Y),neighbors(Y,[C1,C2,C3,C4])] -
            [neg(pit(C1)),neg(pit(C2)),neg(pit(C3)),neg(pit(C4))]
          ]).
```

If the agent senses a breeze, i.e. `breeze(true)`, then we add the information that there might be a pit in each of the four neighboring cells `[C1,C2,C3,C4]` to the agent's current state knowledge. Otherwise we include for each of the neighboring cells that it does not contain a pit.[3]

[2]The figure is courtesy of M. Thielscher

[3]For ease of modeling we assume each cell has four neighbors — the cells on the border do neither contain pits, nor the Wumpus.

We also use the Wumpus world to evaluate the performance of ALPprolog against that of full Flux: The Wumpus world admits a model that is supported by both languages. In [Thielscher, 2005b] a modeling of the Wumpus world in Flux has been presented. Our modeling is very similar, the only difference being that we abstract from the direction the agent is facing: We thus do not have to perform "turn"-actions, the agent can move to any adjacent cell. Once the agent knows in which neighboring cell the Wumpus is located she shoots the arrow in the right direction — also without turning in the right direction.

We use two different models: In one we include all the connections between different cells (i.e. expressions of the form $\text{Connected}(\text{cell}_1, \text{cell}_2)$) into the agent's state knowledge — this serves the purpose of seeing how ALPprolog and Flux handle large state representations. But we can also treat the connections as auxiliary predicate defined in a background theory — this model is easier to handle for both ALPprolog and Flux. We call the first model the "big" Wumpus world, and the second the "small" one.

In table 4.1 we show the respective run-times for some small solvable Wumpus worlds, and the overall number of actions required to achieve the goal. In principle it is hard to ensure that the agent will use the same sequence of actions to achieve a goal, no matter whether the underlying language is ALPprolog or Flux. For our model of the Wumpus world, however, it turned out that initially sorting the state representation (and the connections) is enough to ensure that the same course of action is taken in both settings.

Size	ALPprolog	Flux	Actions
4×4	0.06 s	0.77 s	26
8×8	0.57 s	20.63 s	96
16×16	36.89 s	622.04 s ($\approx$ 10 m)	372
32×32	1655.48 s ($\approx$ 27.5 m)	30836.21 s ($\approx$ 8.5 h)	1559

Table 4.1: Run-times for the Wumpus World — Small Model.

We can make several observations: On this model of the Wumpus world ALPprolog is about twenty times faster than Flux for the 16×16 and the 32×32 settings; for the 8×8 the factor is more than thirty. In this model the memory requirements of ALPprolog and Flux were very similar. For the biggest models both languages used less than 100 MB.

In table 4.2 we show the respective results for the same solvable Wumpus worlds, using the "big" model.

We can make similar observations: Overall, ALPprolog is even more efficient than Flux when the big Wumpus world model is used. Moreover it scales about as well

Size	ALPprolog	Flux	Actions
4×4	0.07 s	2.11 s	26
8×8	1.14 s	101.51 s	96
16×16	55.59 s	4807.37 s ($\approx$ 80 m)	372
32×32	2582.83 s ($\approx$ 43 m)	307421.21s ($\approx$ 3.5 d)	1559

Table 4.2: Run-times for the Wumpus World — Big Model.

as Flux if we look at the transition from the 8×8 world to the 16×16 world. Plan length increases by a factor of about four when moving to a bigger world, and reasoning time increases by a factor of about fifty. The amount of memory needed by Flux and ALPprolog to solve these Wumpus worlds does not differ much: for the bigger worlds about one hundred megabytes are used. For the 32×32 world ALPprolog needs about 120 MB, whereas Flux uses up to 300 MB. In ALPprolog the run-time again increases by a factor of about fifty, whereas for Flux it increases by a factor of about sixty-five. On this model, Flux takes longer than ALPprolog to solve the problem by roughly two orders of magnitude.

Overall, ALPprolog is more efficient at solving this essentially propositional model of the Wumpus world. Both Flux and ALPprolog scale about equally well if we look at the size of the playing field. But for Flux it is more difficult to cope with big state representations. This shows when we compare the increase of the run-time between the small and the big model of Wumpus worlds of the same size: For ALPprolog the run-time is increased by a factor of about two independent from the size of the world, for Flux it is increased by a factor between five (8×8) and ten (32×32).

In [Thielscher, 2005a] the performance of Flux has thoroughly been evaluated against that of Golog. The scenario used for this evaluation featured a robot situated in an office environment, that is supposed to clean the offices, but not to disturb the workers. To this end, the robot can sense whether one of the adjacent offices is occupied. The characteristics of this scenario are sufficiently similar to those of the Wumpus world to transfer some of the observations. First of all, on this cleanbot domain Flux performed consistently better than Golog. Moreover, and as for Special Flux, the performance of full Flux gains continuously over that of Golog the more actions have to be performed. Since ALPprolog scales about as well as Flux, and runs faster, these observations also apply to ALPprolog.

In [Thielscher, 2005a] it has been conjectured that the performance of Flux is superior to that of Golog because of two things: First, Flux uses progression instead of the regression used by Golog. Second it has been argued that the constraint solving in Flux is linear, whereas reasoning with prime implicates in Golog is NP-complete. The performance of ALPprolog, however, seems to imply that this second argument

does not hold. For the same dynamic domain reasoning with prime implicates in ALPprolog does not seem to scale worse than reasoning with the constraints of Flux.

Let us in this context also point out the following: If the formula representing the agent's state knowledge is built on ground literals only (as in our model for the Wumpus world), then the Flux constraint solver transforms it into a representation that contains exactly the prime implicates of the formula. Thus for the Wumpus world both Flux and ALPprolog essentially use the same state representation.

4.7.3 Extension to Offline Planning

It is straightforward to extend ALPprolog so that it can be used to compute unconditional plans: We simply allow backtracking over actions in a derivation, and include an uninstantiated `do(A)` literal in some program clause.

For conditional planning, however, something more is needed: We need to implement a mechanism that computes a most general disjunctive substitution on the action variable `A` in the literal `do(A)`. For this we have to find a set of action preconditions ϕ_i such that both

- $\bigvee_i \phi_i$, and
- no proper subset of these action preconditions

is entailed by the current state. If the action domain features sensors then we will have to implement epistemically adequate planning as discussed in section 3.6.2: We allow disjunctive substitutions only on sense fluents, not on action variables, making the task of finding a suitable disjunctive substitution Θ easier.

Once we have found such a disjunctive substitution Θ we are faced with the reasoning by cases that the CLP(D) proof calculus employs. This can be done by testing whether each of the substitutions $\theta \in \Theta$ together with the respective augmented action domain results in a successful derivation. We leave the actual implementation of this approach to epistemically adequate planning as future work.

If we allow for disjunctive substitutions on action variables in ALPprolog then there is no reason why we should not allow them on object variables, too. However, this will make the implementation of reasoning with the prime implicates more expensive: Assume a PI-list `[[f(a),f(b)]]` together with a query `?- ?(f(X)).`. In the current implementation of ALPprolog this query will simply fail — we are looking for ordinary substitutions only. If we want to extend ALPprolog to offline planning we have to change this behavior — but this will make reasoning more expensive.

4.7.4 Application in General Game Playing

General Game Playing [Genesereth et al., 2005] is a recent AI research area aiming at the integration of various AI research strands: A program (also called a player) is given an axiomatization of the rules of a game. This game may be single- or multi-player. The player then computes a strategy/heuristic that it uses to play and hopefully win the game.

The main challenge of General Game Playing consists of constructing a suitable heuristics. However, the player also needs a means to represent and reason about the state of the game, and the changes effected by the players moves. One successful player is Fluxplayer [Schiffel and Thielscher, 2007] which, as the name suggests, uses Flux for this task. However, up to now all the games played in General Game Playing have been restricted to complete information — there is no need for disjunctive knowledge or open world semantics [Love et al., 2008]. Accordingly, Fluxplayer is based on Special Flux. However, there is currently work going on to broaden the class of games that can be described in the Game Description Language to incomplete and disjunctive information. Hence, ALPprolog might prove useful for a future version of Fluxplayer.

4.7.5 Availability of ALPprolog

The source code for ALPprolog together with the modeling of the Wumpus world can be obtained online at `http://alpprolog.sourceforge.net/`. ECLiPSe Prolog can be obtained from `http://www.eclipse-clp.org/`.

5 Action Logic Programs Based on Description Logic

In this chapter we describe an implementation of a fragment of the ALP framework for the online control of agents where the reasoning about actions is based on Description Logic reasoning, and in particular on the ABox updates first introduced in [Liu et al., 2006].[1] To this end we introduce a fragment of the Fluent Calculus in the UAC where

- the initial state is specified by a DL ABox;
- upon encountering a `do`-literal in the derivation the current state description is progressed via ABox update algorithms; and
- the evaluation of `?`-literals is based on DL consistency checking/query answering.

Moreover, domain constraints can be specified by acyclic TBoxes, and action effects are described by ABoxes, too.

Such an implementation covers exciting new territory for implemented action languages: Most existing implementations are propositional and/or based on the closed world assumption. In contrast, by using ABox updates for reasoning about the action domain we obtain both open world semantics and expressivity that goes considerably beyond classical propositional logic.

This chapter is organized as follows: In section 5.1 we define what Description Logic based action domains in the UAC look like. For this we use the following idea: In [Drescher and Thielscher, 2007] we have given a semantics for ABox update in an old version of the Fluent Calculus — any ABox update in the sense of [Liu et al., 2006] can equivalently be expressed as an effect axiom in this Fluent Calculus. In section 5.1 below we give a version of this result where we identify a fragment of the UAC (and also of the Fluent Calculus expressed in the UAC) for which the ABox update algorithms from [Liu et al., 2006] can be used as a progression mechanism for the initial state. This result allows to use an implementation of ABox update as a reasoning method in the ALP framework.

In section 5.2 we look at the tasks that have to be solved in order to implement the ALP framework atop of logical ABox update: In a first step we show how to exploit query answering in section 5.2.1.

[1] We shall also refer to these ABox updates as logical updates.

When implementing ABox update it quickly became clear that a direct implementation of the respective algorithms from [Liu et al., 2006] (as recapitulated in section 2.4.2) is unworkable. Hence the subsequent section 5.2.2 is devoted to presenting the optimizations we have used in our implementation, and we assess the resulting performance in section 5.2.3. In general, logical ABox updates can only be expressed in Description Logics that cannot directly be handled by current DL reasoners. In section 5.2.4 we present some reasoning methods that do work on updated ABoxes. We evaluate their performance in 5.2.5.

In a final section 5.3 we draw some conclusions wrt. the implementation of the ALP framework atop of Description Logic based action domains.

5.1 ABox Update in the Unifying Action Calculus

In this section we introduce a fragment of the UAC where reasoning is reduced to DL reasoning and to the computation of ABox updates in the sense of [Liu et al., 2006]. The fragment is parametric in the underlying DL — the only requirement according to [Liu et al., 2006] is that the respective DL admits ABox updates based on the possible models approach of [Winslett, 1988]. Hence, in the following we only speak of an ABox, Boolean ABox, TBox, etc. without referring to a particular DL.

5.1.1 Correspondence between FO and State Formulas

We start with a simple technical observation: There is a satisfiability-preserving mapping from first order sentences to state formulas of the UAC. This mapping will subsequently allow us to treat DL formulas as shorthand for the formulas used to axiomatize action domains in the UAC.

This mapping is based on the following observations: Clearly, for any first order language $\mathcal{L}$ with equality we can define a unifying action calculus signature that contains exactly one function symbol F_i of sort FLUENT for every predicate symbol $P_i \in \mathcal{L}$ (except equality). Moreover, its terms $\vec{t}$ of sort OBJECT are precisely the terms of $\mathcal{L}$. Next define the mapping from first order sentences to UAC state formulas:

Definition 5.1 (Mapping $\mathcal{FOL}$ Sentences to State Formulas)**.** *The mapping τ_1 takes first order sentences ϕ in $\mathcal{L}$ to state formulas $\phi(s)$, by replacing every occurrence of an atom $P_i(\vec{t})$ in ϕ with $Holds(F_i(\vec{t}), s)$, where s is a variable of sort* TIME*. The mapping is extended to sets of first order sentences by mapping them to the conjunction of their elements.*

It is not hard to see that an arbitrary FOL sentence ϕ is consistent/satisfiable if and only if $\tau_1(\phi)$ is. Hence, and since DLs can be regarded as fragments of first order logic, we have obtained a correspondence between DL ABoxes, TBoxes, etc., and UAC state formulas $\phi[s]$ uniform in the free time-variable s.

5.1.2 ABox Update Action Domains

We next show how ABoxes, TBoxes, and ABox updates can be used as shorthand for action domains in the UAC. First, recall that an action domain D consists of

- an initial state axiom $\mathsf{D}_{\mathsf{Init}}$,
- a set of precondition axioms $\mathsf{D}_{\mathsf{Poss}}$,
- a set of effect axioms $\mathsf{D}_{\mathsf{Effects}}$,
- a set of domain constraints D_{dc}, and
- foundational axioms $\mathsf{D}_{\mathsf{aux}}$.

All these can also be specified using ABoxes, TBoxes, and ABox updates by exploiting the following observations:

- The domain constraints D_{dc} can be specified by a TBox $\mathcal{T}$: Let $\phi(s) = \tau_1(\mathcal{T})$, then $\forall s \phi(s)$ is the corresponding domain constraint.
- The initial state $\mathsf{D}_{\mathsf{Init}}$ can be specified by an ABox $\mathcal{A}$: in the corresponding state formula $\phi[s] = \tau_1(\mathcal{A})$ we instantiate s to S_0. Without loss of generality we assume that the ABox $\mathsf{D}_{\mathsf{Init}}$ does not contain concepts defined in the TBox.
- Foundational axioms $\mathsf{D}_{\mathsf{aux}}$ as usual contain an axiomatization of the situations, and unique name axioms for the actions that we introduce below. But recall that ABox update has to be parametrized by a set $\mathcal{EQ}$ of (dis-)equalities between individual names (cf. section 2.4.2). Accordingly the corresponding (dis-)equality axioms are included in the foundational axioms $\mathsf{D}_{\mathsf{aux}}$.

It is worth pointing out that acyclic TBoxes allow a natural model of sensor axioms. For example, we can write

$$SF \equiv \{v_1\} \sqcap C_1 \sqcup \ldots \sqcup \{v_n\} \sqcap C_n,$$

to specify the meaning of the n different possible sensing results v_n.

Next we turn to the effect axioms, and the respective precondition axioms. First recall that a conditional ABox update is a finite set of pairs ϕ/ψ, where the assertion ϕ is the condition part and the literal ψ the conditional effect. Concepts defined in the TBox may be used in the condition part ϕ, but not in the effect ψ. Assume given a conditional update $\mathcal{U}$: First we associate with every such update an action constant Action. The precondition of each of these actions can again be specified by an ABox $\mathcal{A}$ that corresponds to the UAC precondition axiom

$$\mathrm{Poss}(\mathrm{Action}, s_1, s_2) \equiv \tau_1(\mathcal{A}) \wedge s_2 = \mathrm{Do}(\mathrm{Action}, s_1).$$

We now turn to the construction of an effect axiom corresponding to the update $\mathcal{U} = \{\phi_1/\psi_1, \ldots, \phi_n/\psi_n\}$. Define the set of conditional effects $\mathcal{CE}_1 = \{\phi_j/\psi_j \mid \phi_j/\psi_j \in \mathcal{U}\} \cup \{\neg\phi_j/\text{nil} \mid \phi_j/\psi_j \in \mathcal{U}\}$ and let $\mathcal{CE}_2$ be the set of all subsets of $\mathcal{CE}_1$ that are maximally consistent (in the propositional sense) with regard to the condition part ϕ_j. Note that the cardinality of $\mathcal{CE}_2$ will be exponential in the size of $\mathcal{U}$. Intuitively the set $\mathcal{CE}_2$ represents all the possible cases of a conditional update. Next we stipulate that the corresponding effect axiom in the UAC be of the form

$$\begin{aligned}&\text{Poss}(A, s_1, s_2) \supset \\ &\bigvee_k (\Phi_k[s_1] \wedge (\forall f)[(\bigvee_i f = f_{ki} \vee (\text{Holds}(f, s_1) \wedge \bigwedge_j f \neq g_{kj})) \qquad (*) \\ &\qquad\qquad \equiv \text{Holds}(f, s_2)]),\end{aligned}$$

where

- k ranges over all the possible cases $\mathcal{CE}_2$;
- $\Phi_k[s_1]$ corresponds to the condition part of the respective case; and
- all the positive and negative effects (f_{ki} and g_{ki}) are ground fluents, corresponding to the effects of the case (unless the effect is nil).

So for each conditional update $\mathcal{U}$ from $\mathcal{CE}_2$ (i.e. a case) in the UAC effect axiom we set Φ to the UAC representation of the condition part of $\mathcal{U}$, i.e. $\bigwedge\{\tau_1(\phi) \mid \phi/\psi \in \mathcal{U}\}$. For the effects we first make sure that those effects mentioned by $\mathcal{U}$ take place. But in addition we also have to ensure that the effects respect the TBox/domain constraints. Overall this is achieved as follows:

- First set $\delta_j^+[s_1]'$ (or $\delta_j^-[s_1]'$, respectively) to the disjunction of the negative effects (or the positive effects, respectively) of $\mathcal{U}$, i.e. set, e.g., $\delta_j^-[s_1]'$ to

$$\bigvee_{\{\neg\psi \mid \phi/\psi \in \mathcal{U} \text{ and } \psi \neq \text{nil}\}} f = \tau_2(\psi),$$

  where the mapping τ_2 maps FO/DL literals $(\neg)P_i(\vec{t})$ to the corresponding UAC terms $F_i(\vec{t})$.

- Then extend these preliminary positive and negative effects as follows, where C is any concept name defined in the TBox $\mathcal{T}$and $\mathcal{A}$ is the "effect ABox" $\bigwedge\{\psi \mid \phi/\psi \in \mathcal{U} \text{ and } \psi \neq \text{nil}\}$:
  - Add to $\delta_j^+[s_1]'$ all fluents $\tau_2(C(a))$ where $\mathcal{A} \cup \mathcal{T} \models C(a)$.
  - Add to $\delta_j^-[s_1]'$ all fluents $\tau_2(C(a))$ where $\mathcal{A} \cup \mathcal{T} \models \neg C(a)$.

Observe that all the different cases $\Phi[s_1]$ in the effect axiom are mutually exclusive. In particular the effect axioms are deterministic in the sense of definition 3.8. Observe also that the form of the effect axiom satisfies the conditions that identify Fluent Calculus effect axioms in the UAC. This finishes the construction of an effect axiom corresponding to a conditional ABox update.

Let us summarize how an action domain based on ABox updates is to be represented in the UAC:

Definition 5.2 (ABox Update Action Domain)**.** *An action domain in the UAC that is based on DL ABox update (henceforth denoted D_{DL}) is specified by the following:*

- *an ABox $\mathcal{A}$, as shorthand for the initial state formula;*
- *a set $\mathcal{EQ}$ of (dis-)equalities between individual names;*
- *a finite set of action pairs $< \mathcal{A}_i, \mathcal{U}_i >$, where $\mathcal{A}_i$ is an ABox, and $\mathcal{U}_i$ an ABox update; and*
- *an acyclic TBox $\mathcal{T}$, as shorthand for the domain constraints.*

5.1.3 UAC Semantics for ABox Update

We still need to show that the mapping from conditional updates to UAC effect axioms is correct. The following proposition tells us that this indeed is the case:

Proposition 5.1 (UAC Semantics for ABox Update)**.** *Let an action domain D_{DL} be given. Moreover, let $\mathcal{U}_i$ be a conditional update such that $\mathsf{D}_{DL} \models Poss(Action_i, s_1, s_2)$. For any model $\mathcal{I}$ of D_{DL}, the relation between the set of fluents that hold at s_1 in $\mathcal{I}$ and the set of fluents that hold at s_2 in $\mathcal{I}$ and the relation between an original interpretation and an updated interpretation of an ABox (cf. definition 2.16) coincide.*

Sketch. The proof is based on two observations: First, in the respective effect axiom $\Psi[s_1, s_2]$ of the form $(*)$ exactly one case $\eta_j[s_1, s_2]$ is satisfied by $\mathcal{I}$. Second, the effect axiom correctly captures the semantics of ABox update because it axiomatizes the relationship between what held prior to the actions execution and what holds after its execution as

$$(\forall f)\,[\delta_j^+[s_1] \vee (\mathrm{Holds}(f, s_1) \wedge \neg\delta_j^-[s_1]) \equiv Holds(f, s_2)],$$

where $\delta_j^+[s_1]$ and $\delta_j^-[s_1]$ enumerate the positive and negative effects. □

5.1.4 Modularity of ABox Update Action Domains

As discussed in section 2.3.4 the modularity of action domains ensures that there is no "bad" interaction between the different types of axioms. In particular, this is important for the implementation of action programming languages because reasoning about the action domain can be reduced to reasoning about the appropriate axioms instead of the complete axiomatization. Fortunately, modularity holds for ABox update action domains under a simple condition:

Proposition 5.2 (Modularity of ABox Update Action Domains)**.** *If the equality theory $\mathcal{EQ}$ underlying an ABox update action domain D_{DL} is the unique name assumption then D_{DL} is modular.*

The construction of effect axioms corresponding to ABox updates ensures that whenever an action is possible then at least one of the effect cases applies. The same construction also ensures that no implicit effects or preconditions are entailed by the TBox. If all objects have unique names the underlying equality theory cannot cause implicit effects either. For our purposes the unique name assumption is not a restriction at all since for ALPs we already stipulate that all objects have unique names.

5.2 Implementing ALPs atop ABox Update

In this section we describe our work on implementing LP(D_{DL}) and CLP(D_{DL}) — (constraint) logic programming over action domains that are based on ABox update. The implementation consists of three components:

- For the evaluation of program atoms we can use existing mature Prolog technology.
- HOLDS atoms are evaluated against the ABox describing the respective world state.[2]
- The evaluation of Poss atoms consists of two steps: First we evaluate the action's precondition against the current ABox. If the precondition is satisfied we update the current ABox with the action's effects to a new ABox and make this the current ABox.

Recall that defined concept names in action domains D_{DL} may only be used in action preconditions, and the condition of a conditional effect, but not in the initial ABox, or an effect itself. Because in addition ABox update action domains are modular we can take the following approach to reasoning:

[2] Due to the leftmost computation rule of Prolog, and as in ALPprolog, for this we can always use the "current" ABox.

- Initially we verify that the initial ABox $\mathcal{A}$ does not contradict the domain constraints (TBox $\mathcal{T}$).
- We evaluate action preconditions, state properties from the ALP, and conditions of effects wrt. the knowledge base $(\mathcal{A}, \mathcal{T})$.
- We do not add the action's effects concerning defined concept names to the updated ABox $\mathcal{A}'$. It is safe to simply evaluate queries that mention defined concept names against the knowledge base $(\mathcal{A}', \mathcal{T})$.

Recall that all the actions in Poss atoms are constants in D_{DL}. If we make the additional, natural assumption that the state properties in HOLDS atoms are ABoxes then in the evaluation of the special atoms there are no substitutions at all, and hence we can use the LP(D) proof calculus.

However, for applications this approach may prove a hindrance: Assume we are interested in an instance of the concept C. Instead of forcing the programmer to check for every individual name i in the domain whether $C(i)$ is entailed by the current ABox, we want to be able to ask whether there is some individual i such that the current ABox entails $C(x)\{(x/i)\}$: In short, we want to exploit query answering. A similar argument can be made for the usefulness of parametric actions. However, DL based action domains are unlikely to be query-complete: expressing disjunctive and existential information is a strong-point of DLs. Hence, we cannot use the LP(D) calculus for a complete implementation: For this we have to resort to the CLP(D) proof calculus instead. If we are only interested in the online execution of programs we can stick with the LP(D) calculus.

The rest of this section is organized as follows:

- First, we generalize ABox update action domains to query answering.
- Next we describe implementation techniques that proved useful in the computation of updated ABoxes.[3]
- Then we describe techniques that can be used to reason with updated ABoxes — standard DL reasoners will not do because updated ABoxes may contain the @-constructor, complex roles or Boolean combinations of assertions.

5.2.1 Query Answering for Action Domains D_{DL}

As discussed above, from a practical perspective, exploiting query answering techniques is desirable. However, the alert reader will have noticed that the above definition 5.2 of an ABox update action domain D_{DL} does not feature parametric

[3] Some of this material has been presented in [Drescher et al., 2009].

actions. In this section we generalize the action domains D_{DL} accordingly. To this end we first introduce ABox patterns:

Definition 5.3 (ABox Pattern)**.** *An assertion pattern is an expression of the form $r(x,y)$, $r(x,i)$, $r(i,x)$ or $C(x)$ where x, y are variables. An ABox pattern $\mathcal{A}(\vec{x})$ is a finite set of ABox assertions or assertion patterns, where $\vec{x}$ denotes all the variables from the assertion patterns. An ABox update pattern is a set of pairs $\phi(\vec{x})/\psi(\vec{x})$, where $\phi(\vec{x})$ is an assertion pattern, and $\psi(\vec{x})$ is a literal assertion pattern. Finally, an ABox action pattern is a term $Action_i(\vec{x})$ where $\vec{x}$ is a vector of variables.*

Next we generalize the definition of an ABox update action domain D_{DL} by including ABox patterns and parametric actions:

Definition 5.4 (ABox Update Action Domain — New Version)**.** *Henceforth, by D_{DL} we denote an action domain in the UAC that is specified by the following:*

- *a finite set of action triples $< Action_i(\vec{x}), \mathcal{A}_i(\vec{x}), \mathcal{U}_i(\vec{y}) >$, where*
  - *$Action_i(\vec{x})$ is an ABox action pattern;*
  - *$\mathcal{A}_i(\vec{x})$ is an ABox pattern (the action precondition); and*
  - *$\mathcal{U}_i(\vec{y})$ is an ABox update pattern, where $\vec{y}$ is contained in $\vec{x}$.*[4]
- *an ABox $\mathcal{A}$(the initial state), a set $\mathcal{EQ}$ of (dis-)equalities between individual names (the underlying equality theory), and an acyclic TBox $\mathcal{T}$ (the domain constraints) — all as before.*

Let us point out that the UAC semantics for ABox update is readily generalized to this notion of D_{DL} by speaking of a ground instance of an ABox action pattern in proposition 5.1.

In ALPs built atop D_{DL}, ABox patterns will serve as the state properties in HOLDS-atoms, just as ABox action patterns will be used by Poss-atoms. However, for this definition of an ABox update action domain D_{DL}, we may be confronted with actions that are applicable, yet we cannot compute the resulting updated ABox. Consider the following example:

Example 5.1 (Variables in ABox Update). Let the initial state be specified by $\mathcal{A} = \{a : \exists r.C\}$ and consider an action triple

$$< \mathrm{Action}(x), \{r(a,x), C(x)\}, \{\neg C(x)\} > .$$

Clearly, the action precondition $(\exists x) r(a,x) \wedge C(x)$ is entailed by $\mathcal{A}$. However, ABox update cannot cope with variable effects.

[4] This restriction ensures that for an applicable ground action we can also compute the updated ABox.

Hence, in an implementation of ALPs atop D_{DL} we restrict the substitution rule of the CLP(D) calculus to ground, but possibly disjunctive, substitutions. The constraint rule is restricted to the evaluation of ABoxes, not ABox patterns, for similar reasons. These two restrictions identify the fragment of CLP(D) that an implementation based on ABox update can maximally cover.

In practice it is even advisable to restrict the CLP(D) to ground, non-disjunctive substitutions: We have not found a sufficiently efficient method for obtaining the disjunctive substitutions that the CLP(D) proof calculus calls for (cf. the discussion in section 5.2.4 below).

This is especially sad because the notion of a disjunctive substitution together with the CLP(D) calculus constitute a neat generalization of ABox update to disjunctive effects: Consider, e.g., an ALP atom $\mathrm{Poss}(A(\vec{x}), s_1, s_2)$ such that $\mathsf{D}_{\mathrm{DL}} \models \mathrm{Poss}(A(\vec{x}), s_1, s_2)\Theta$ with disjunctive substitution Θ. The reasoning-by-cases approach taken in the CLP(D) approach now ensures that the deterministic ABox update algorithm that we implemented suffices. Hence we can do conditional planning even if the ABox update algorithms do not support disjunctive effects.

If we further refrain from using the constraint rule then essentially we get sound derivations in LP(D) over an action domain that probably is not query-complete. Then, however, our implementation cannot be used for conditional, but only for conformant planning.

5.2.2 Implementing ABox Update

Our implementation of ABox update is based on the respective constructions recapitulated in section 2.4.2. Hence, our implementation supports the DLs $\mathcal{ALCO}^{@}$ and $\mathcal{ALCIO}^{@}$, or $\mathcal{ALCO}^{+}$ and $\mathcal{ALCIO}^{+}$, respectively. In principle the construction of updated ABoxes from [Liu et al., 2006] also works on the extension of these DLs with qualified number restrictions, that is on $\mathcal{ALCQIO}^{@}$ and $\mathcal{ALCQIO}^{+}$. With qualified number restrictions we can, e.g., express that some individual violates the one-child-policy of China in the form of the following assertion:[5]

$$\text{someone} :\geq 2\ \text{hasChild}.\top.$$

But as before, in the following we will not address the case of qualified number restrictions. The reason for this is twofold:

- For $\mathcal{ALCQIO}^{@}$ the construction of updated qualified number restrictions involves a lot of case distinctions (cf. [Liu et al., 2006]), and hence results in updated ABoxes so huge that they cannot be managed in practice.

[5] We assume marital fidelity, and ignore the issue of multiples.

- For $\mathcal{ALCQIO}^+$ in turn we could not find a reasoner that directly supports both qualified number restrictions (or, equivalently, counting quantifiers) and the Boolean role constructors. If we compile away the qualified number restrictions to first order logic with equality we face two difficulties: The result of the compilation is outside of the two variable fragment with equality that is known to be decidable [Pratt-Hartmann, 2005] — we need more variables. Hence decision procedures for $\mathcal{FOL}^2$ cannot directly be applied. Moreover, the compilation substantially increases the formula size.

As a matter of fact, our implementation does support the update of ABoxes with qualified number restrictions. However, the resulting ABoxes cannot be handled in practice for reasons given above, and hence, in this thesis, we did not recall the respective material.

Our prototypical implementation of ABox update also covers only unconditional updates. This restriction is motivated by the observation that it is already difficult to control the non-determinism present in unconditional ABox updates. By this latter non-determinism we mean the following case distinction for unnamed objects in the ABox: Is the object affected by the update, or not?

Moreover the usage of the unique name assumption as underlying equality theory for ABoxes is mandatory if we want to build ALPs on top of ABox update, because we chose to avoid equational unification in our proof calculi (cf. section 3.3).

Next we outline techniques that we have found useful for the implementation of ABox update. We start with a simplification of the definition of an updated ABox. Recall, that the updated ABox $\mathcal{A}'$ is defined as (cf. proposition 2.2):

$$\mathcal{A}' \;=\; \bigvee_{\mathcal{D}\in\mathfrak{D}} \bigwedge \mathcal{A}^{\mathcal{D}_\mathcal{U}} \cup \mathcal{D}_\mathcal{U} \cup (\mathcal{D} \setminus \mathcal{D}_\mathcal{U}^{\neg}). \tag{5.1}$$

Next consider the following construction of a new ABox $\mathcal{B}$ from the input ABox $\mathcal{A}$, and the update $\mathcal{U}$:

$$\mathcal{B} \;=\; \bigvee_{\mathcal{D}_\mathcal{U}\subseteq\mathcal{U}} \bigwedge \mathcal{A}^{\mathcal{D}_\mathcal{U}} \cup \mathcal{U}. \tag{5.2}$$

We can show the following proposition:

Proposition 5.3 (Updated ABox)**.** *Let $\mathcal{A}'$ be constructed according to equation (5.1), and let $\mathcal{B}$ be obtained using equation (5.2). Then $\mathcal{A}'$ and $\mathcal{B}$ are equivalent.*

Updating Boolean ABoxes

Updating an ABox according to Proposition 2.2 or 2.3 results in a Boolean ABox. In [Liu et al., 2006] this updated ABox is transformed to a non-Boolean ABox using the @-constructor, before it is updated again. The following observation shows that

Boolean ABoxes can directly be updated again by updating the individual assertions, avoiding the transformation.

Observation 5.1 (Distributivity of Update)**.** *ABox update distributes over the logical connectives conjunction and disjunction in Boolean ABoxes; i.e.*

$$(\mathcal{A}_1 \boxtimes \mathcal{A}_2) * \mathcal{U} \equiv (\mathcal{A}_1 * \mathcal{U}) \boxtimes (\mathcal{A}_2 * \mathcal{U}),$$

where $\boxtimes$ *denotes either* $\wedge$ *or* $\vee$ *(negation can be pushed inside the assertions).*

By updating a Boolean ABox directly we also obtain a slightly more compact representation than the original one — the update $\mathcal{U}$ is no longer contained in two disjuncts:

Observation 5.2 (Updating Boolean ABoxes)**.** *For a Boolean ABox* $\mathcal{A}$*(we assume negation has been pushed inside the assertions), let the updated ABox* $\mathcal{A}'$ *be defined as*

$$\mathcal{A}' = (\mathcal{A} \circledast \mathcal{U}) \wedge \bigwedge \mathcal{U}.$$

Here $\mathcal{A} \circledast \mathcal{U}$ *is defined recursively as*

$$\begin{aligned} \alpha \circledast \mathcal{U} &= \bigvee_{\mathcal{D}_\mathcal{U} \subseteq \mathcal{U}} \alpha^{\mathcal{D}_\mathcal{U}} \\ (\mathcal{A} \boxtimes \mathcal{B}) \circledast \mathcal{U} &= (\mathcal{A} \circledast \mathcal{U}) \boxtimes (\mathcal{B} \circledast \mathcal{U}) \end{aligned}$$

where $\boxtimes$ *denotes* $\wedge$ *or* $\vee$*,* α *is an assertion, and* $\alpha^{\mathcal{D}_\mathcal{U}}$ *is defined as in Proposition 2.2 (or 2.3) for* $\mathcal{ALCO}^+$ *(or for* $\mathcal{ALCO}^{@}$*, respectively). Then* $\mathcal{A} * \mathcal{U} \equiv \mathcal{A}'$*.*

As already stated this construction avoids the duplication of the update itself. But it also results in a different formula type for the updated, Boolean ABox. Let us first explain our notion of Boolean ABoxes in CNF and DNF: By replacing every assertion in a Boolean ABox $\mathcal{A}$ with a propositional letter we obtain a propositional formula $F_\mathcal{A}$. Now we say that the ABox $\mathcal{A}$ is a *Boolean ABox in CNF (resp. DNF)* if $F_\mathcal{A}$ is in CNF (resp. DNF). Following the above construction the updated ABox is in CNF, whereas the original construction produces DNF ABoxes. Whereas in the original construction ABoxes were updated as a whole, the above construction updates ABoxes assertion-by-assertion.

In order not to clutter up the notation in the following we will use $\mathcal{D}$ instead of $\mathcal{D}_\mathcal{U}$ to denote an arbitrary subset of an update $\mathcal{U}$. We will also call such a subset $\mathcal{D}$ a diagram.

Determinate Updates

Looking at the construction of updated ABoxes, we see that for an ABox $\mathcal{A}$, and an update $\mathcal{U}$ in the updated ABox we get a disjunct for every subset (diagram) $\mathcal{D}$ of $\mathcal{U}$. This causes a rapid growth of the updated ABox. If, however, $\mathcal{D}$ contains a literal δ such that either δ or its negation is entailed by the ABox $\mathcal{A}$, then some of the computed updated disjuncts will be inconsistent and can hence be removed:

Observation 5.3 (Determinate Updates)**.** *For Boolean ABox $\mathcal{A}$, update $\mathcal{U}$, assertion $\delta \in \mathcal{U}$, and $\mathcal{D}$ a diagram of $\mathcal{U}$ we have that $\bigwedge \mathcal{U} \cup \mathcal{A}^{\mathcal{D}}$ is unsatisfiable if either*

- *$\mathcal{A} \models \delta$ and $\delta \in \mathcal{D}$; or*
- *$\mathcal{A} \models \neg\delta$ and $\delta \notin \mathcal{D}$.*[6]

Otherwise, if $\mathcal{A} \not\models \delta$ and $\mathcal{A} \not\models \neg\delta$, for all $\delta \in \mathcal{D}$, then $\mathcal{A}^{\mathcal{D}}$ is consistent.

Detecting this type of situation requires up to two reasoning steps for every literal δ in the update: $\mathcal{A} \models \delta$ and $\mathcal{A} \models \neg\delta$, resulting in a trade-off between time and space efficiency: Spending more time on reasoning may result in a smaller updated ABoxes.

Exploiting the Unique Name Assumption

The mandatory unique name assumption (UNA) enforces that no two individual names may denote the same object. The constructions from Proposition 2.2 and 2.3 do not take the UNA into account, however; in our implementation we construct simpler updated ABoxes by keeping track of the individuals $\vec{s}$ and $\vec{t}$ that an assertion $\gamma(\vec{s})$ refers to when updating it with $\delta(\vec{t})$:

Definition 5.5 (Updated Assertion with UNA)**.** *We define the ABox $\mathcal{A}^{\mathcal{D}}$ as $\mathcal{A}^{\mathcal{D}} = \{C^{\mathcal{D},a}(a) \mid C(a) \in \mathcal{A}\} \cup \{r^{\mathcal{D},a-b}(a,b) \mid r(a,b) \in \mathcal{A}\}$. The interesting part of the definition of $C^{\mathcal{D},i}$ and $r^{\mathcal{D},i-j}$ is given in figure 5.1 — we omit the straightforward Boolean constructors. For $\mathcal{ALCO}^{@}$ ABoxes we only use the modified construction of concept assertions.*

Let us illustrate how the modified construction is working by means of a small example:

Example 5.2 (Exploiting UNA). If we update the ABox $\mathcal{A} = \{A(i)\}$ with $\mathcal{D} = \{\neg A(j)\}$, we can easily obtain $A(i)$, instead of $A \sqcup \{j\}(i)$ using the standard construction. But next consider the ABox $\mathcal{A} = \{\forall r.(\{j\} \sqcap A)(i)\}$, updated by $\mathcal{D} = \{A(k)\}$. As part of the standard update construction we obtain $\forall r.(\{j\} \sqcap (A \sqcap \neg\{k\}))(i)$ which can be simplified using UNA to $\forall r.(\{j\} \sqcap A)(i)$. In our implementation we cannot detect this latter case, which can only be identified by reasoning.

[6] The latter of these two observations can already be found in [Liu et al., 2006].

$A^{\mathcal{D},i} =$	$\top$, if $\neg A(i) \in \mathcal{D}$	$A^{\mathcal{D},i} =$	$\bot$, if $A(i) \in \mathcal{D}$
$A^{\mathcal{D},i} =$	A, if $\{(\neg)A(i)\} \cap \mathcal{D} = \emptyset$		
$r^{\mathcal{D},i-j} =$	$\top$, if $\neg r(i,j) \in \mathcal{D}$	$r^{\mathcal{D},i-j} =$	$\bot$, if $r(i,j) \in \mathcal{D}$
$r^{\mathcal{D},i-j} =$	r, if $\{(\neg)r(i,j)\} \cap \mathcal{D} = \emptyset$		
$\{i\}^{\mathcal{D},i} =$	$\top$	$\{i\}^{\mathcal{D},j} =$	$\bot$
$\{(i,j)\}^{\mathcal{D},i-j} =$	$\top$	$\{(i,j)\}^{\mathcal{D},k-l} =$	$\bot$, if $k \neq i$ or $l \neq j$
$(\exists r.C)^{\mathcal{D},i} =$	$\exists r.(C^{\mathcal{D}})$, if $q(i,x) \notin \mathcal{D}$ for $q \in \mathsf{sub}(r)$	$(\forall r.C)^{\mathcal{D},i} =$	$\forall r.(C^{\mathcal{D}})$, if $q(i,x) \notin \mathcal{D}$ for $q \in \mathsf{sub}(r)$
$(\exists r.C)^{\mathcal{D},i} =$	$(\exists r.C)^{\mathcal{D}}$, otherwise	$(\forall r.C)^{\mathcal{D},i} =$	$(\forall r.C)^{\mathcal{D}}$, otherwise
$(@_j C)^{\mathcal{D},i} =$	$@_j C^{\mathcal{D},j}$	$(@_i C)^{\mathcal{D}} =$	$@_i C^{\mathcal{D},i}$

Figure 5.1: Constructing $C^{\mathcal{D},i}$ and $r^{\mathcal{D},i-j}$ for $\mathcal{ALCO}^+$ and $\mathcal{ALCO}^@$

On the one hand, this UNA-based construction is not costly at all. On the other hand, it cannot identify all cases where the UNA admits a more concise updated ABox.

Omitting Subsuming Disjuncts and Entailed Assertions

Intuitively, in a disjunction we can omit the "stronger" of two disjuncts: Let the disjunction $(\mathcal{A}^{\mathcal{D}_1} \vee \mathcal{A}^{\mathcal{D}_2})$ be part of an updated ABox. If $\mathcal{A}^{\mathcal{D}_1} \models \mathcal{A}^{\mathcal{D}_2}$ then $(\mathcal{A}^{\mathcal{D}_1} \vee \mathcal{A}^{\mathcal{D}_2}) \equiv \mathcal{A}^{\mathcal{D}_2}$. Detecting subsuming disjuncts in general requires reasoning. But by a simple, syntactic check we can detect beforehand some cases where one of the disjuncts $\mathcal{A}^{\mathcal{D}_1}$ and $\mathcal{A}^{\mathcal{D}_2}$ will subsume the other. Then the computation of subsuming disjuncts can be avoided.

We say that an occurrence of a concept or role name δ in an assertion is *positive*, if it is in the scope of an even number of negation signs, and *negative* otherwise.

Observation 5.4 (Detecting Subsuming Disjuncts)**.** *For an ABox $\mathcal{A}$ that is to be updated with update $\mathcal{U}$ it holds that:*

(1) if the update $\mathcal{U}$ is positive wrt. the concept or role name δ then
- *if δ occurs only positively in $\mathcal{A}$ then $\mathcal{A}^{\mathcal{D}_1} \models \mathcal{A}^{\mathcal{D}_2}$; and*
- *if δ occurs only negatively in $\mathcal{A}$ then $\mathcal{A}^{\mathcal{D}_2} \models \mathcal{A}^{\mathcal{D}_1}$,*

for all subsets $\mathcal{D}_1, \mathcal{D}_2$ of $\mathcal{U}$ such that $\mathcal{D}_1 = \mathcal{D}_2 \setminus \{\delta(\vec{t}) \mid \delta(\vec{t}) \in \mathcal{U}\}$.

(2) if the update $\mathcal{U}$ is negative wrt. the concept or role name δ then
- *if δ occurs only positively in $\mathcal{A}$ then $\mathcal{A}^{\mathcal{D}_2} \models \mathcal{A}^{\mathcal{D}_1}$; and*
- *if δ occurs only negatively in $\mathcal{A}$ then $\mathcal{A}^{\mathcal{D}_1} \models \mathcal{A}^{\mathcal{D}_2}$,*

again for all subsets $\mathcal{D}_1, \mathcal{D}_2$ of $\mathcal{U}$ such that $\mathcal{D}_1 = \mathcal{D}_2 \setminus \{\delta(\vec{t}) \mid \delta(\vec{t}) \in \mathcal{U}\}$.

This means that in case (1) we retain only those diagrams that do not contain any of the $\delta(\vec{t})$ assertions, whereas in case (2) we retain only those diagrams that contain all the $\delta(\vec{t})$ assertions.

Conversely, we can also avoid updating entailed assertions:

Observation 5.5 (Omitting Entailed Assertions)**.** *Let $\mathcal{A}$ be an ABox and $\mathcal{U}$ an update. If $\mathcal{U} \models \alpha$ or $\mathcal{A}\backslash\{\alpha\} \models \alpha$ for some assertion $\alpha \in \mathcal{A}$, then $\mathcal{A}*\mathcal{U} \equiv (\mathcal{A}\backslash\{\alpha\})*\mathcal{U}$.*

Removing all entailed assertions might be too expensive in practice; one might try doing this periodically.

Independent Assertions

Next we address the question under which conditions an assertion in an ABox is not affected by an update. We say that assertion α in an ABox $\mathcal{A}$ is independent from update $\mathcal{U} = \{\delta\}$ iff $\mathcal{A} * \mathcal{U} \equiv \alpha \wedge (\mathcal{B} * \mathcal{U})$ where $\mathcal{B} = \mathcal{A} \setminus \{\alpha\}$. The more independent assertions we can identify, the more compact our ABox representation becomes.

Detecting this in all cases requires reasoning steps and thus is costly. It is easy, though, to syntactically detect some of the independent assertions:

Observation 5.6 (Independent Assertion)**.** *The assertion $\alpha(\vec{t_2})$ is independent from the update $\mathcal{U}$ for an ABox $\mathcal{A}$ in negation normal form if for all concept and role names δ such that $(\neg)\delta(\vec{t_1}) \in \mathcal{U}$*

- *if $\delta \notin \mathsf{sub}(\alpha)$, or*
- *if $\mathcal{A} \vDash \vec{t_1} \neq \vec{t_2}$, δ occurs in α only outside the scope of a quantifier, and for all subconcepts $@_i C$ of α the assertion $C(i)$ is independent from $\mathcal{U}$.*

Assertion Representation

For various reasons it is useful to define a normal form for assertions: In our implementation we move negation, disjunction, and quantifiers as far inward into the assertions as possible. We also do some lexicographic ordering, and split conjunctions outside of the scope of quantifiers. This results in a non-canonical normal form, that is both cheap to compute, and where many equivalent assertions are represented syntactically identical. Similar techniques are standard in DL reasoning; for us this will prove particularly useful in the section 5.2.4 on DPLL(T) below, where we assign propositional variables to assertions.

Our normal form ensures that ABoxes are always in negation normal form (NNF); in our implementation we also use a variant of the update construction that from ABoxes in NNF directly produces updated ABoxes in NNF.

5.2.3 Evaluation of ABox Update Implementation

In this section we provide some empirical results wrt. the efficacy of the various optimization techniques introduced above.

For the evaluation we use our own implementation of logical ABox update in ECLiPSe-Prolog [ECLiPSe Implementors Group, 2009], and an implementation of projective ABox updates (cf. section 2.4.2) by Hongkai Liu in Java.[7] The testing was done in joint work with Hongkai Liu, who kindly also provided the testing data. Testing was done using a set of 2000 random $\mathcal{ALC}$ ABoxes containing between two and twenty-three assertions, and a set of 2000 random, unconditional updates containing between one and eleven effects. The methodology underlying the generation of the random input is described in [Liu, 2009].

To begin with, figure 5.2 (courtesy of Hongkai Liu) shows the sizes of updates based on projective updates (plotted as $\times$) and a naive implementation of logical updates in $\mathcal{ALCO}^{@}$ (plotted as $+$), where by a naive implementation of logical updates we refer to an implementation that is directly based on the construction of logical updates as recapitulated in section 2.4.2 — with no optimizations whatsoever. The horizontal axis shows the size of the input ABox, and the vertical axis shows the logarithmized size of the output ABox ($ln(n)$). Figure 5.2 seems to confirm the theoretical expectation that projective updates (being polynomial in the input) behave much better than logical updates, which are exponential in all of the input.

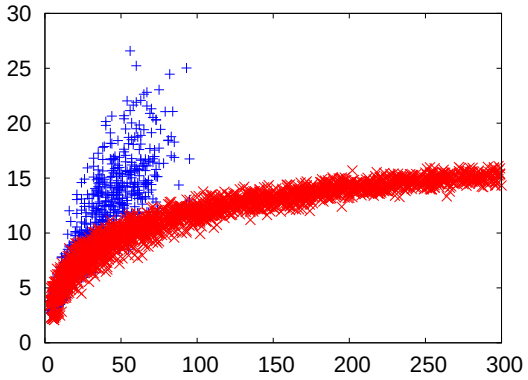
Let us next discuss the efficacy of the various optimization techniques for logical updates introduced above.

We start with an observation on updating ABoxes to either CNF or DNF. The other optimizations notwithstanding, updating to CNF due to the duplication of identical assertions results in updated ABoxes that are significantly bigger than those obtained by updating to DNF. As expected in the experiments the CNF representation proved far superior.

For all the other optimization techniques the bottom-line is that the syntactic variants outperform their counterparts involving reasoning. For example, in the 2000 random input ABoxes, together with the 2000 random updates we could not find a single assertion that was independent from the update by resorting to reasoning for which we could not detect this syntactically. The costs for the syntactic fragments of the optimization techniques proved to be negligible, and on the random input data all of the optimization resulted in smaller updated ABoxes. Hence, our implementation syntactically does the following:

- Transform ABoxes into a certain normal form;
- Exploit the unique name assumption;

[7] For clarity, in this section we refer to ABox update as logical update to distinguish it from projective update.



+ : Logical updates in $\mathcal{ALCO}^{@}$

× : Projective updates

The numbers in vertical axes are logarithmized ($ln(n)$).

Figure 5.2: Comparing the size of projective and logical updates

- Update directly to negation normal form;
- Detect subsuming disjuncts; and
- Identify assertions that are independent from the update.

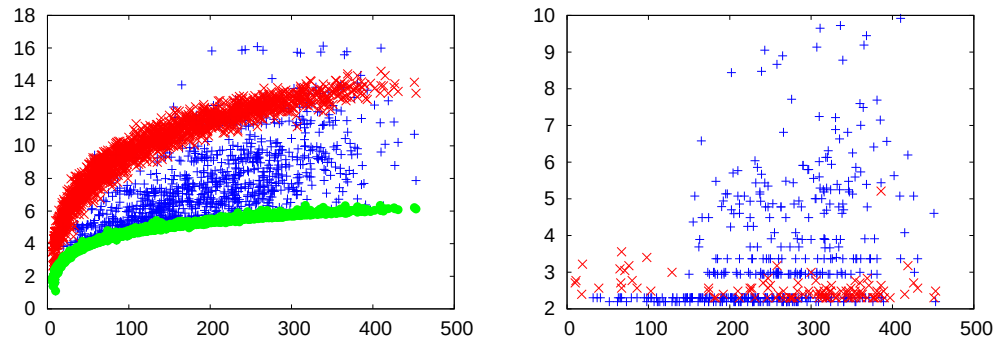
Regarding optimizations that require reasoning we could make the following observations:

- The costs for detecting and removing entailed assertions were prohibitively high. We have not looked into heuristics for identifying suitable candidate assertions, though.
- With regard to the identification of determinate updates we could make an interesting observation: On the set of the 2000 random input data the syntactic technique for identifying subsuming disjuncts was able to detect virtually all determinate updates.[8]

[8]This means that there is no assertion α in the testing data such that a concept or role name occurs both negatively and positively in α.

Summing up, neither of the two optimization ideas based on reasoning proved worth the cost.

With this we come to the comparison of our optimized implementation of logical updates to the implementation of projective updates on the same 2000 testing data. In figure 5.3 (also courtesy of Hongkai Liu) we can see that the optimized implementation works well on the random testing data. The left graph shows the size of the updated ABoxes for all 2000 testing data, while the graph on the right shows the respective sizes of those updated ABoxes that are of size up to 400 000. The sizes of projective updates are plotted as ×, whereas • and + denote the sizes of logical updates in $\mathcal{ALCO}^+$ and $\mathcal{ALCO}^@$, respectively. Here, the horizontal axis shows the size of the input ABox, and the vertical axis shows the logarithmized size of the output ABox in the left graph, and the logarithmized run-time in milliseconds in the right graph — both times ($ln(n)$). We do not show the computation time for updated ABoxes in $\mathcal{ALCO}^+$ (•) in the right graph — it is almost identical to that of projective updates (×).



\+ : Logical updates in $\mathcal{ALCO}^@$

× : Projective updates

• : Logical updates in $\mathcal{ALCO}^+$

The numbers in vertical axes are logarithmized ($ln(n)$).

Figure 5.3: The size of projective and optimized logical updates

We can make the following observations: In general, the logical updates in $\mathcal{ALCO}^+$ are the smallest. Next the logical updates in $\mathcal{ALCO}^@$ frequently behave better than the projective updates — however, there is a number of cases where their theoretical inferiority shows up. Overall, the behavior of projective updates is more predictable

than that of logical updates in $\mathcal{ALCO}^{@}$. For the latter, all observed "bad" cases occurred if the update contained assertions that concerned subconcepts and -roles that occurred at a quantifier depth greater than one in the input ABox — i.e. the update concerned unnamed individuals.

The following numbers also highlight the differences between the competing representations: The random input data consume about 17 MB of disk space. The updated ABoxes consume about 20 MB in $\mathcal{ALCO}^{+}$, about 1.1 GB in $\mathcal{ALCO}^{@}$, and about 1.5 GB as projective updates. To be fair, in $\mathcal{ALCO}^{@}$ not even all updated ABoxes could be computed — there are about sixty ABoxes that are not included in the 1.1 GB but in the 1.5 GB of the projective updates. However, the most striking fact is the superiority of $\mathcal{ALCO}^{+}$.

This somewhat surprising superiority of the exponential logical updates in $\mathcal{ALCO}^{+}$ over the polynomial projective updates can at least in part be attributed to the fact that none of the optimization techniques introduced above can directly be applied to projective updates. Developing powerful optimization techniques for projective updates is thus an interesting topic for future work.

Let us also briefly comment on the issue of iterated updates: The results in figures 5.2 and 5.3 are for non-iterative updates. As discussed in [Liu, 2009] projective updates do not handle iterated updates well in practice. One way around this is to merge iterated updates into a single one [Liu, 2009], e.g., to merge the updates $\mathcal{U}_1 = \{A(a)\}$ and $\mathcal{U}_2 = \{\neg A(a)\}$ into $\mathcal{U} = \{\neg A(a)\}$. While for projective updates this technique is vital, for $\mathcal{ALCO}^{+}$ it is not yet clear whether it is cheaper to repeatedly enforce the same effect or to accept a small increase in redundancy of the updated ABox.

Summing up, as of now logical updates in $\mathcal{ALCO}^{+}$ appear to be the best representation formalism for ABox update.

5.2.4 Reasoning with Updated ABoxes

In this section we discuss the different reasoning methods that we have used for reasoning with updated ABoxes. Basically, we distinguish two reasoning types:

- If the respective HOLDS-atom from the ALP is ground it is sufficient if the reasoner decides ABox (un-)satisfiability.
- If the HOLDS-query is non-ground, i.e. contains variables, the reasoner has to support query answering.

Theoretically, the latter reasoning task can be reduced to the former, by iteratively trying instantiations of the query with all the individual names of the domain. Note that this theoretical possibility of answering queries by checking ground query

entailment extends to disjunctive substitutions; we simply generate candidate disjunctions. Here the practical infeasibility of the approach is especially apparent. In practice dedicated support for query answering thus is an important distinguishing feature of the different reasoning methods.

Another distinguishing feature of the reasoning methods is the logic supported, i.e., whether the reasoner supports $\mathcal{ALCO}^{@}$ or $\mathcal{ALCO}^{+}$. Incidentally, the fact that updated $\mathcal{ALCO}$ ABoxes are in either $\mathcal{ALCO}^{@}$ or $\mathcal{ALCO}^{+}$ is one of the key challenges of reasoning with updated ABoxes: State-of-the-art DL reasoners like Pellet, FaCT++, and Racer, support neither logic. The other key challenge is that these reasoners also do not support Boolean ABox reasoning. In the following we will present five different approaches for reasoning with updated ABoxes that in principle can cope with these challenges:

- one which is based on propositional satisfiability testing modulo theories — the DPLL(T) approach;
- one which uses a consistency preserving reduction from a Boolean ABox to a non-Boolean ABox;
- one which uses Spartacus, a tableau prover for hybrid logic;
- one which uses Otter, a first-order theorem prover; and
- one which uses MetTeL, a tableau reasoner that supports Boolean ABoxes.

The first three of these approaches work for $\mathcal{ALCO}^{@}$, while the latter two work for $\mathcal{ALCO}^{+}$. Only the reduction approach and Otter provide dedicated support for query answering. It is worth pointing out that we cannot expect to get an implementation of CLP(D_{DL}) for conditional planning unless we use a theorem prover: The notion of a disjunctive substitution (or a disjunctive answer to a query) is not supported by all other reasoners.

For the first two approaches we have to deal with the @ constructor. Using the equivalence-preserving, exponential transformation from [Liu et al., 2006] for compiling the @ constructor away we can obtain an equivalent Boolean ABox. However, if we instead simulate the @-operator by a universal role [Bong, 2007] we obtain a linear consistency-preserving transformation — we adopt this approach.

In our implementations we use Pellet as a DL reasoner because it supports nominals, query answering and pinpointing [Sirin et al., 2007].

DPLL(T)

The work on the DPLL(T) approach was conducted by Hongkai Liu. Here we only give a brief outline of the approach, the details can be found in [Liu, 2009].

Most of the contemporary state-of-the-art SAT-solvers [Een and Sörensson, 2003, de Moura and Bjørner, 2008] implement variants of the well-known Davis-Putnam-Logemann-Loveland (DPLL) procedure for deciding the satisfiability of clauses in propositional logic [Davis and Putnam, 1960, Davis et al., 1962]. The DPLL(T) approach combines a DPLL procedure with a theory solver that can handle conjunctions of literals in the theory to solve the satisfiability problem modulo theories (SMT) [Nieuwenhuis et al., 2007]. In DPLL(T) a DPLL procedure works on the propositional formula obtained by replacing the theory atoms with propositional letters. Whenever the DPLL procedure extends the current partial interpretation by a new element the theory solver is invoked to check consistency of the conjunction of the theory atoms corresponding to the partial, propositional interpretation. If the theory solver reports an inconsistency, the DPLL procedure will back-jump and thus the search space is pruned. The consistency problem of Boolean ABoxes becomes an instance of SMT if we take ABox assertions as theory atoms and a DL reasoner as theory solver.

In DLs the inference problem of explaining which (combinations of) assertions in an ABox cause the ABox to be inconsistent is known as pinpointing [Schlobach, 2003, Baader and Peñaloza, 2008]. In the DPLL(T) approach these explanations can then be used to build back-jump clauses [Nieuwenhuis et al., 2007].

Hongkai Liu implemented the DPLL(T) approach with the strategy of MiniSat [Een and Sörensson, 2003], using Pellet as theory solver because of its support for pinpointing. This implementation is henceforth called Pellet-DPLL. This approach works for $\mathcal{ALCO}^{@}$ and its extensions, but lacks dedicated support for query answering.

Pellet Reduction

We can linearly compile Boolean $\mathcal{ALCO}^{@}$ ABoxes to classical $\mathcal{ALCO}^{@}$ ABoxes [Liu et al., 2006]. Then, simulating the @-operator by a universal role, we can directly use a standard DL reasoner; this approach is henceforth called Pellet-UR. This approach works for $\mathcal{ALCO}^{@}$ and its extensions, and directly supports query answering.[9]

Automated Theorem Proving: Otter

As we have seen in figure 5.3 the DL $\mathcal{ALCO}^{+}$ admits small updated ABoxes. However, its role operators are not supported by current mature DL reasoners. By translating $\mathcal{ALCO}^{+}$ to first order logic [Borgida, 1996] we can use theorem provers that naturally can cope with Boolean role constructors (and also Boolean ABoxes).

[9] The idea for this approach originates from Carsten Lutz.

We chose to use Otter [McCune, 2003] because it supports query answering via answer literals [Green, 1969]. After a few experiments we chose to configure Otter to use hyper-resolution combined with Knuth-Bendix-rewriting, plus the set-of-support strategy.

MetTeL

MetTeL is a tableau based reasoner that supports a variety of logics; among them is the DL $\mathcal{ALBO}$ [Schmidt and Tishkovsky, 2007] that features Boolean role operators. While $\mathcal{ALBO}$ does not directly support the nominal role of $\mathcal{ALCO}^+$, the nominal role can still be expressed in $\mathcal{ALBO}$. In particular we have used the following encoding of the nominal role: We introduce a reserved role name u and for every pair of individual names (a, b) a fresh role name rab — rab denotes the nominal role $\{(a, b)\}$. We then include the following axiom in our domain description:

$$\forall u \sqcup \neg u.[(\forall rab.\{b\}) \sqcap (\forall \neg rab.\neg\{b\}) \sqcap (\forall rab^{-}.\{a\}) \sqcap (\neg rab^{-}.\neg\{a\})].^{10}$$

Intuitively, this axioms says that for all y reachable via the universal role $u \sqcup \neg u$ the following holds: If y is connected via rab to another individual then that individual is $\{b\}$, and if there is no connection via rab to some other individual then this individual is $\neg\{b\}$. The second half of the axiom is analogous. Accordingly $rab(c, d)$ iff $\{a\}(c)$ and $\{b\}(d)$. There is another similar encoding of the nominal role in MetTeL using $\mathcal{ALBO}$'s domain and range restrictions for roles. A strong-point of MetTeL is that it directly supports Boolean ABox reasoning. However, it has no built-in support for query answering.

Spartacus

In his thesis Hongkai Liu has also evaluated the performance of the reasoner Spartacus [Götzmann, 2009], a dedicated reasoner for hybrid logic. $\mathcal{ALCO}^{@}$ can be regarded as a notational variant of hybrid logic [Liu, 2009], and hence Spartacus is also applicable to reasoning with updated ABoxes. Moreover, Spartacus directly supports Boolean ABoxes. However, Spartacus, only decides satisfiability of (Boolean) ABoxes: It lacks dedicated support for query answering.

5.2.5 Evaluation of Reasoning with Updated ABoxes

In this section we evaluate how well the various proposed reasoning methods worked. In particular, we address the following questions:

- Which reasoning method performs best if the underlying logic is $\mathcal{ALCO}^{@}$ (or, respectively, $\mathcal{ALCO}^{+}$)?

[10] MetTeL accepts this expression as an axiom.

- And how do $\mathcal{ALCO}^{@}$ and $\mathcal{ALCO}^{+}$ compare with regard to reasoning?
- How do the logical updates we implemented compare to the theoretically tractable projective updates?
- How does the ABox update approach compare to a regression-based DL action formalism wrt. reasoning?
- How does reasoning with ABox updates compare to the propositional approach of ALPprolog?
- Finally, how important is dedicated support for query answering?

Testing Scenarios

We have evaluated the performance of the above methods for reasoning in two different settings: We have used more randomly generated testing data, and we have investigated a simple DL version of the Wumpus world (cf. section 4.7.2). The testing results were obtained in joint work with Hongkai Liu.

The random testing data were created by combining the 2000 random input ABoxes and updates with 20000 randomly generated $\mathcal{ALCO}$ ABoxes serving as queries.[11] The random queries have again kindly been provided by Hongkai Liu; information on the parameters used to generate the testing data can be found in his thesis [Liu, 2009].

We have used the random testing data in two different scenarios:

- In one testing scenario we applied only a single update to an input ABox, and then performed a reasoning step.
- For the second, more realistic scenario, we assume an agent is endowed with an initial world model. It then performs some reasoning steps, next it performs an action, updates its world model, again performs some reasoning steps, and so on.

In the second testing scenario we have extended the Wumpus world to a DL problem by adding the (redundant) assertion $\exists \mathrm{at}.\top(\mathrm{wumpus})$.

Reasoning in $\mathcal{ALCO}^{@}$ and $\mathcal{ALCO}^{+}$

We start by comparing reasoning with updated $\mathcal{ALCO}^{@}$ ABoxes to reasoning with updated $\mathcal{ALCO}^{+}$ ABoxes. We used both the random testing data, and the Wumpus world to evaluate ABox consistency checking, and query answering.

[11] Without a query we can never detect inconsistency, because updated ABoxes always are consistent.

Reasoning in $\mathcal{ALCO}^{@}$ For reasoning with updated $\mathcal{ALCO}^{@}$ ABoxes we could make the following observations: For consistency checking of ABoxes in CNF, Spartacus, Pellet-DPLL, and Pellet-UR exhibited a comparable behavior. For query answering the clear winner is Pellet-UR.

Spartacus performed best by a small margin for ABox consistency checking. Pellet-DPLL generally did better than Pellet-UR for detecting an actual inconsistency, while it performed worse than Pellet-UR if the ABox was consistent with the negated query: The overhead introduced by doing DPLL on top of DL reasoning really pays in the case of an inconsistent ABox. Here, the inevitable clashes result in back-jump clauses that help to prune the search space. For a consistent ABox, in the extreme case the DL tableau reasoner might be able to construct a model without encountering any clashes. In this case the redundancy of the additional construction of a propositional model via DPLL is especially apparent. Hence it may be a fruitful approach to build a hybrid reasoner that interleaves these two approaches.

If query answering is among the reasoning tasks, then Pellet-UR is to be preferred over both Pellet-DPLL and Spartacus because of Pellet's direct support for this inference.

For ABox consistency checking the biggest problems occurred if the update was relevant to unnamed individuals in the original ABox, because in the case of $\mathcal{ALCO}^{@}$ this leads to a dramatic increase in ABox size (cf. the discussion in section 5.2.3) above.

The performance of the DPLL(T) approach depends on the performance of the SAT solver and the pinpointing service. Thus Pellet-DPLL could benefit from a more efficient implementation of these tasks.

However, realistic application scenarios involve query answering, and this is a task at which only Pellet-UR excels. For Pellet-DPLL and Spartacus to be applicable it would be vital to first develop heuristics for finding suitable individual names as well as other optimizations for query answering.

Reasoning in $\mathcal{ALCO}^{+}$ For reasoning in $\mathcal{ALCO}^{+}$ we have tried a resolution based theorem prover (Otter) and a tableau prover (MetTeL).

Using Otter as a theorem prover might be considered somewhat unfair to the theorem proving approach, since it is no longer actively maintained and optimized. We chose to use Otter because it supports query answering, which is not supported by most current provers [Waldinger, 2007], but vital in some domains. At the time of writing, efforts were under way to reestablish query answering as a theorem proving task (cf. `www.cs.miami.edu/~tptp/TPTP/Proposals/AnswerExtraction.html`). There is likewise work going on to make theorem provers more effective on large, but simple axiom sets. Thus it may be possible to resort to state-of-the art theorem provers for reasoning in $\mathcal{ALCO}^{+}$ in the near future.

In general, the behavior of Otter was not predictable, i.e. the reasoning times varied greatly. The bottleneck appeared to be the conversion from ABoxes in CNF to full first order CNF. State-of-the-art resolution provers use different transformation algorithms [Nonnengart and Weidenbach, 2001] that result in more manageable clause sets.

The performance of MetTeL was more predictable than that of Otter, although in some cases it seems to depend on a random seed used to ensure fair tableau derivations. Moreover, and contrary to Otter, MetTeL is a decision procedure for $\mathcal{ALCO}^+$. Although, $\mathcal{ALCO}^+$ can in principle also be decided by resolution, in practice resolution theorem provers often discard clauses they no longer deem necessary, and this behavior may cause them to be incomplete.

For MetTeL the biggest problems appeared to be that it is not optimized for ABox reasoning at all, and that it does not use back-jumping to prune the search space. The usage of back-jumping is generally considered a key component in the implementation of an efficient DL reasoner [Baader et al., 2003]. The encoding of the nominal role via a universal role introduces a lot of avoidable non-determinism in the derivations, too.

Comparing $\mathcal{ALCO}^@$ and $\mathcal{ALCO}^+$ The most important result wrt. reasoning with ABoxes updated to $\mathcal{ALCO}^+$ instead of reasoning with their counterparts updated to $\mathcal{ALCO}^@$ is that it is not competitive. Pellet-UR on top of $\mathcal{ALCO}^@$ proved to perform better in almost all of the conducted tests. This may be somewhat surprising, given that $\mathcal{ALCO}^+$ has proven to be a far superior representation language for updated ABoxes (cf. section 5.2.3 above).

Of course theoretically reasoning in $\mathcal{ALCO}^+$ is NExpTime-complete compared to the PSpace completeness of $\mathcal{ALCO}^@$. This jump in complexity is due to the fact that in $\mathcal{ALCO}^+$ we can use complicated role constructions inside quantifier restrictions. Starting from the same original $\mathcal{ALCO}$ ABox and then updating it to either a Boolean ABox in $\mathcal{ALCO}^+$ or $\mathcal{ALCO}^@$, however, at least intuitively the structure of the $\mathcal{ALCO}^+$ ABox is far simpler. However, our experiments clearly showed that $\mathcal{ALCO}^@$ together with Pellet-UR is superior to $\mathcal{ALCO}^+$ with either Otter or MetTeL.

At this point we would like to conjecture that it should be possible to build a tableau prover similar to MetTeL that

- fully supports Boolean ABox reasoning;
- exploits back-jumping;
- supports query answering;

- has dedicated support for the nominal role;[12] and
- is competitive with state-of-the-art DL reasoners.

This may finally allow us to exploit the fact that $\mathcal{ALCO}^+$ admits smaller updated ABoxes than $\mathcal{ALCO}^@$. Moreover, we believe that this combination of a hypothetical reasoner with $\mathcal{ALCO}^+$ as underlying DL should allow to fully exploit the potential of ABox update.

We conjecture that a tableau based reasoner is more likely to result in good performance than a resolution based theorem prover for the following reasons: Theorem provers are — as the name suggests — typically designed to prove theorems. Hence their performance often is poor if the clause set is satisfiable. Moreover, they are typically designed to find proofs from a small number of intricate axioms. We expect that in application scenarios support for a big number of fairly simple assertions is more important: Recall that ABox update works best if the update does not concern unnamed, i.e. quantified individuals. Some discussion of the issue tableau based versus resolution based reasoning in Description Logics can be found in, e.g., [Tsarkov and Horrocks, 2003] and [Horrocks and Voronkov, 2006] — although this discussion is chiefly concerned with TBox reasoning some of the raised issues apply to ABox reasoning, too.

A strength of the theorem prover approach is that it naturally supports the notion of a disjunctive answer substitution via answer literals — for tableau based reasoners this reasoning task to the best of our knowledge has not been studied yet. Of course, a resolution prover tailored for DL and in particular also ABox reasoning may be a viable alternative. There is, e.g., some recent work on deciding the DL $\mathcal{SHOIQ}$ via resolution, where the authors emphasize that they aim also at efficient ABox reasoning with a future version of the reasoner KAON2 [Kazakov and Motik, 2008].

Let us contemplate this issue from one more angle: Recently, it has been argued by Raphael Volz in his dissertation [Volz, 2007] that neither tableau calculi nor classical theorem proving techniques can match the needs of efficient DL ABox reasoning. Instead he proposed to use techniques from deductive database technology for this task. The resulting system is reasonably efficient, albeit incomplete. Statistically it appears to have a high recall. If we were to try this approach the major obstacle is that the proposed techniques cannot cope with nominals yet — and for constructing updated ABoxes these are indispensible.

Logical and Projective Updates versus Regression in DLs

In the previous section we have seen that for logical updates the combination of $\mathcal{ALCO}^@$ as representation language and Pellet-UR as reasoner worked best. In this

[12] Extending the tableau for $\mathcal{ALBO}$ [Schmidt and Tishkovsky, 2007] to support the nominal rule poses no major difficulties.

section we compare logical updates, projective updates and a form of DL based regression. The reasoning task we studied is inspired by the following realistic scenario: An agent that is endowed with an initial world model will most likely interleave reasoning steps with the execution of actions that affect the world model.

We have used the same testing data, and this testing, too, has been done in joint work with Hongkai Liu. The task we used for testing consisted of alternatingly first posing one query, and then applying an update, starting from a random initial ABox.

Table 5.1 shows run-time results for selected ABoxes: We picked two small, middle-sized, and big input ABoxes each, where for one of these the final output ABox was relatively small, and for the other relatively big (ABoxes 37, 406, 713, 1124, 1331, 1712).[13] The numbers indicate how many update-and-reason steps could be completed in a thirty minute time-frame. If all 2000 steps have been completed the approximate run-time is shown in minutes in parentheses.

Hongkai Liu has also implemented the DL based action formalism presented in [Baader et al., 2005]. It has been shown in [Milicic, 2008] that this action formalism can be seen as a variant of regression-based reasoning in the Situation Calculus [Reiter, 2001a]. We have used this implementation to compare regression to progression (i.e. ABox update) in DL based action formalisms. In table 5.1 the regression implementation is shown under the name "Projection".

ABox	Pellet-UR	Projective Update	Projection
37	2000 (8.8 min)	2000 (5.1 min)	2000 (12.2 min)
406	2000 (9.7 min)	2000 (5.4 min)	2000 (14.1 min)
713	168	80	2000 (13.9 min)
1124	78	6	2000 (15.1 min)
1331	39	11	238
1712	19	1	181

Table 5.1: Reasoning with sequences of updates.

From this table we can see that logical updates resulted in better performance than projective updates, especially if the problems got harder. However, we can also see that the same holds of the regression implementation which performed best on the hardest problems. For these hardest problems, the logical updates were more likely to "blow up" at some point — by this we mean an update that results in a dramatic increase in ABox size. However, also the regression approach suffers from occasionally dramatic increases in knowledge base size. Even though projective updates are the only approach not affected by similar "explosions" they exhibited even worse performance than logical updates.

[13] Sizes refer to optimized logical update.

Comparing ABox Update to Propositional Reasoning

In this section we have a look at the price we have to pay for the expressivity of DL ABoxes, as compared to propositional state representation. To do this we extend the simple propositional model of the Wumpus world presented in section 4.7.2 to a DL problem by adding the (redundant) assertion $\exists \mathrm{at}.\top(\mathrm{wumpus})$. The model only had to be slightly adapted, because in DLs we do not have nested terms at our disposal: We write, e.g., `cell11` instead of `cell(1,1)`. Some results for this scenario are shown in table 5.2, where PL1 denotes the propositional model introduced in section 4.7.2, and DL1 is the respective DL extension. Unsupported expressivity is denoted by by n/a in table 5.2.

Model	ALPprolog	Pellet-UR
4x4 PL1	0.01 s	21.4 s (-4.6 s)
8x8 PL1	0.2 s	1024.3 s (-525.5 s)
4x4 DL1	n/a	19.9 s (-4.7 s)
8x8 DL1	n/a	984.6 s (-522.3 s)

Table 5.2: Run-times for the DL Wumpus World.

The first thing we note is that for Pellet the presence of a single quantifying assertion is insignificant — in fact, if any effect can be noticed at all it is a positive one. Next we have to note that there is a huge gap wrt. performance between the propositional and the DL reasoner.

This may be attributed to several factors: First of all, the library we use for interfacing Prolog to Pellet is inefficiently implemented — in parentheses we show the time the program spends on the task of Pellet input generation. But even if we disregard the respective overhead the difference in runtime is huge. On the one hand, ALPprolog and Pellet use a different path through the Wumpus world, because of the different order in which substitutions are generated. However, the Wumpus worlds are generated in such a way that an almost complete exploration of the world is performed in both cases. Moreover, our implementation of ABox update ensures that the DL model also maintains the current state as a conjunction of prime implicates, and hence reasoning should not be considerably harder for Pellet than for ALPprolog. Overall we conclude that wrt. reasoning with updated ABoxes there still seems to be plenty of room for improvement.

The Role of Query Answering

We have used the DL version of the Wumpus world and the random testing data to evaluate the importance of dedicated support for query answering. Our experiments

confirmed that it is vital: The more individual names occur in the application domain the more the advantages become apparent. The random testing data featured twenty named individuals, whereas in the Wumpus world there were up to seventy of them. Accordingly, in some of the experiments Otter performed better than Pellet-DPLL, Spartacus, and MetTeL because of its support for query answering via answer literals.

5.3 Perspectives for ALPs Based on Description Logics

In this section we try to assess the perspectives for an implementation of the ALP framework atop of DL based action domains. As we have seen above the experimental results for logical ABox update are mixed, in a rather unfortunate sense:

- If we base ABox update on the DL $\mathcal{ALCO}^+$ we can obtain small updated ABoxes at a low cost. But we could not find a reasoner that works well for $\mathcal{ALCO}^+$.
- State-of-the-art DL reasoners worked fairly well on ABoxes that have been updated to $\mathcal{ALCO}^@$, however. But we have seen that $\mathcal{ALCO}^@$ is a poor choice for representing updated ABoxes.

We have seen that the theoretically tractable projective updates in practice performed worse than the logical updates. They are based on a less straightforward encoding of the occuring changes, and hence it proved hard to find optimization techniques similar to those applicable in the case of logical updates [Liu, 2009].

We have also seen that a regression based implementation performed slightly better than the progression based formalism of ABox update. This contradicts our assumption that progression tends to gain over regression the longer the action sequences considered become.

We have also seen that there is a huge gap in performance between DL reasoning and propositional reasoning on essentially the same modelings of the Wumpus world.

We take all this to show that the implementation of ABox update and reasoning with updated ABoxes does not yet live up to its potential. As discussed above we conjecture that the key ingredient required for obtaining a decent implementation of ALPs atop of Description Logics is a (tableau based) reasoner that is optimized for query answering and reasoning with Boolean ABoxes in $\mathcal{ALCO}^+$ (and its extensions).

6 Relation to Existing Work

In this chapter we explore the relationship between our ALP framework and the manifold existing work on specifying strategies for agents in dynamic domains. In view of both the generality of our framework and the huge body of existing work this is a daunting task. Our own work focuses on the specification of strategic behavior in dynamic domains using classical logic — as motivated in the introduction. We simplify our task by exploring only the relation between our framework and approaches that are likewise based on classical logic. In particular we will discuss how action logic programs relate to the following languages:

- Golog, the imperative, Situation Calculus-based agent control language;
- Flux, the logic programming dialect based on the Fluent Calculus; and

Thus we completely ignore, e.g., the work on agent strategies using

- non-monotonic logic and in particular answer set programming (see, e.g., [Gelfond and Lifschitz, 1992, Baral, 2003] for an introduction); and
- so-called belief-desire-intention architectures for agent programming that are not based on formal logic (see, e.g., [Bordini et al., 2007, Mascardi et al., 2005, Spark Implementors Group, 2009]).

But we do discuss the relation of ALPs to dedicated planners, systems that can be used to solve planning problems in a domain-independent fashion. Likewise we discuss recent theoretical results concerning reasoning about action domains. These theoretical insights pave the way for future implementations of action reasoners that can then be plugged into the ALP framework.

6.1 Golog

Arguably, Golog [Levesque et al., 1997, Reiter, 2001a] is the first action programming language that has been built on top of action theories. It is a very expressive language which is based on Algol-inspired programming constructs for defining agent strategies. In the basic version it features the following constructs: Test actions (testing whether some property holds), primitive actions, sequential execution of actions, non-deterministic choice of arguments and actions, non-deterministic iteration, and

procedures defined in terms of the aforementioned constructs — conditionals as well as while-loops can be defined as syntactic sugar. There are also more advanced versions of Golog that feature concurrent execution of actions, interrupts, and priorities.

Golog programs are usually built atop Situation Calculus action domains — although it is straightforward to use other action formalisms, as long as they are based on the time structure of situations, and the Poss atom. The original logical semantics for Golog programs has been given in the Situation Calculus [Reiter, 2001a]. It is based on the idea of macro-expanding programs to formulas of the Situation Calculus. In order to express reachability of situations, for non-deterministic iteration and procedures second order axioms are used, and hence this logical semantics of basic Golog is already quite intricate.

In this section we discuss the similarities and differences between Golog and ALPs. In particular, we discuss the following:

- What are the differences and commonalities between ALPs and basic Golog?
- Can basic Golog programs be expressed as ALPs?
- Can advanced Golog features be accommodated in the ALP framework?

With regard to practical implementations we will also include a short discussion of the basic Golog interpreter.

6.1.1 The Relation between ALPs and Basic Golog

A first difference between ALPs and basic Golog concerns the "feel" of the language: The former are inspired by declarative logic programming in the style of pure Prolog; the latter language is inspired by imperative programming constructs familiar from Algol.

However, basic Golog and ALPs are quite similar in that both are macro-expanded to sets of classical logical formulas that in turn provide their semantics. In the case of basic Golog the macro-expansion yields a set of first and second order Situation Calculus sentences, which can then be combined with any basic action theory in the same language. For ALPs we obtain a set of first order Horn clauses, which contain atoms from the underlying action theory, but also refer to an additional program signature.

Second Order or First Order Semantics? The major difference between the logical semantics of basic Golog and ALPs is that the former is second order, whereas the latter is first order. Capturing precisely what this difference really amounts to is not easy. But let us point out the following: In [Levesque et al., 1997] the authors give

a Golog interpreter as a Prolog program, i.e., a set of first order Horn clauses. Then they write:

"Given the simplicity of the characterization of the `do` predicate (in first order Horn-clauses), and the complexity of the formula that results from Do (in second order logic), a reasonable question to ask is why we even bother with the latter. The answer is that the definition of `do` is too weak: it is sufficient for finding a terminating situation (when it exists) given an initial one, but it cannot be used to show non-termination. Consider the program $\delta = [a^*; (x \neq x)]$. For this program, we have that $\neg \text{Do}(\delta, s, s')$ is entailed for any s and s'; the `do` predicate, however, would simply run forever."

Here, the construct a^* denotes non-deterministic iteration (zero or more times) of an action. Analogously, we can write the following ALP:

```
p.
p :- do(a), p.
```

Evaluating the query `?- p, ?(X \= X)` is then analogous to evaluating the Golog program δ.

Certainly the authors did not mean to say that the corresponding set of first order clauses is satisfiable. But they are correct to point out that in the SLD-proof calculus (underlying Prolog and ALPs) we obtain an infinite derivation. But this seems to be a failure of the proof calculus rather, than a failure of the first order logical semantics, and hence the question about the necessity of the second order semantics remains unresolved.

On a sidenote, this discussion raises the question why we base ALPs on such a simplistic proof calculus. After all, a more clever proof calculus should be able to detect immediately that the above query will not succeed. This point may be rebutted by paraphrasing [O'Keefe, 1990] where it has been argued that Prolog is an efficient programming language because it is not a clever theorem prover. This amount to saying that, because it is easy to understand what Prolog (or ALPs) will do, the programmer should be able to recognize that the above program plus query does not make sense.

But let us get back to the discussion of the relation between the first order semantics of ALPs and the second order semantics of Golog: The Golog interpreter from [Levesque et al., 1997] is first order (like ALPs), while the Golog semantics is second order. But already the authors of [Levesque et al., 1997] themselves point out that it is a non-trivial task to state in which sense precisely the Golog interpreter is correct. To the best of our knowledge this relation has never been made formally precise. Likewise we were unable to derive a formal result relating the semantics of ALPs and Golog.

Inferring Plans Let us next turn to one of the commonalities of basic Golog and ALPs: Both basic Golog programs and ALPs are macro-expanded to sets of logical formulas, and hence programs are not objects (terms) of the underlying action theory. Because of this it is not possible to quantify over programs, forestalling the program synthesis approach of Manna and Waldinger [Manna and Waldinger, 1987] — this observation is from [Levesque et al., 1997]. However, both Golog and ALPs can be used for planning, and the plans inferred can be regarded as simple programs.

Here, the major practical difference between ALPs and basic Golog is that the former allow to infer conditional plans by appealing to the concept of a disjunctive substitution. In the planning version of Golog [Reiter, 2001a], however, the plans considered are linear. In order to fully exploit the fact that in Situation Calculus action domains a goal holds at some situation if and only if there is a conditional plan achieving the goal [Savelli, 2006], basic Golog at least needs to be modified — perhaps by admitting disjunctive substitutions in the non-deterministic choice operators.

Independence from Particular Action Calculi Another ostensible difference between basic Golog and ALPs is that for the former up to now only semantics in situation-based action calculi exist, while the latter also admit action domains that are based on other time structures. It is hence worth pointing out that Golog can easily be made independent from the Situation Calculus by defining it on top of the UAC, just like the ALPs.

Separation between Strategy and Dynamic Domain Basic Golog and ALPs also differ on the issue of the separation between action domain and agent strategy. Assume we want to define a numeric measure on some fluents that currently hold to guide our strategy — say we want to count, e.g., the number of pawns we still have in the game. In basic Golog this cannot be defined, so the defining formulas have to be already present in the action domain. In ALPs the respective clauses can naturally be formulated as part of the strategy.

6.1.2 Expressing Golog Programs as ALPs

We next show that ALPs provide counterparts for all of the basic Golog constructs, while at the same time they retain a straightforward declarative semantics. The only price we pay is that sometimes we have to introduce auxiliary atoms.

Translating Basic Golog Constructs For all the basic Golog constructs with the exception of procedures the corresponding ALP constructs are summarized in figure 6.1. Golog programs δ are recursively expanded, simultaneously constructing

a corresponding ALP. The expressions δ_1, δ_2 range over Golog subprograms in figure 6.1, and we let `d1`, `d2` range over the rule heads of the respective ALP rules. For every subprogram δ we have to introduce an auxiliary rule head atom `p` — of course this has to be a fresh predicate symbols for every rule.

Golog Construct	**ALP Construct**
Primitive Action a	`Do(a)`
Tests Φ?	`?(Phi)`
Sequence $\delta_1; \delta_2$	`p :- d1, d2.`
Non-deterministic choice of arguments $(\pi x)\delta(x)$	`p :- d(X).`
Non-deterministic choice of actions $\delta_1 \vert \delta_2$	`p :- d1.  p :- d2.`
Non-deterministic iteration δ^*	`p.  p :- d, p.`

Figure 6.1: Golog and ALP constructs

Translating Procedures Golog procedures are defined as expressions of the form **proc** $P(\vec{x})$ δ **endProc**, consisting of a procedure name P, procedure arguments $\vec{x}$, and a procedure body δ that may use all the basic Golog constructs. In particular the procedure body may include procedure calls like, e.g., a recursive call to the procedure P itself. Procedure declarations are translated to ALPs without introducing an auxiliary rule head predicate symbol. Instead for the procedure $P(\vec{x})$ the corresponding rule head will be `p(X)`. The procedure body is expanded using the translation depicted in figure 6.1, only that every procedure call $P(\vec{y})$ is translated to `p(Y)`.

Calling the Program The ALP corresponding to a Golog program δ is called by submitting the auxiliary atom `p` corresponding to δ as a query.

Properties of the Translation We have given ALP analogons of basic Golog programs. Intuitively, we expect that there will be a successful Golog run if and only if the corresponding ALP admits a successful derivation. Stating this formally, however, is not trivial. The difficulties arise because of the second order logical semantics of Golog programs — these are the same difficulties that have so far precluded a formal correctness result for the Golog interpreter. Hence, all we say here is that ALPs provide programming constructs that appear to fit the same needs as the programming constructs available in basic Golog.

Golog Interpreter as ALP From a different viewpoint, we can sidestep the issue of relating the second order semantics of Golog with first order logic programs as

follows: The basic Golog interpreter is a logic program — hence it can be viewed as an ALP. It is further possible to suitably insert the ALP primitives `do` and `?` into the defining clauses of Golog's `do` for the case of `primitive_action` and `holds`. The resulting ALP may then be regarded as a reconciliation of the procedural programming style of Golog with the declarative programming paradigm underlying the ALPs.

6.1.3 Advanced Golog Features and ALPs

The basic version of Golog has seen two main extensions: The first was the extension to ConGolog, a language that features interleaved concurrency and interrupts [Giacomo et al., 2000]. In a second step ConGolog has been extended to IndiGolog, a language that allows to switch between the online and the offline execution of programs [Sardina et al., 2004].

For both these languages, the semantics has been given not by macro-expanding programs to sets of logical formulas, but by an operational state transition semantics. This semantics again has second order features (transitive closure), and moreover requires the encoding of programs as terms in the language of the underlying action theory.[1]

ALPs and Concurrency The lack of support for action domains with concurrent actions is probably the main weakness of the ALP framework. It is also the area where the transition semantics for Golog really shows its strengths. We leave the extension of the ALP framework to concurrent actions, and possibly also interrupts for reactive agents, as a challenge for future work.

Interleaving Online and Offline Execution of Programs The main innovation of IndiGolog is the introduction of a search operator. An IndiGolog program is executed in an online fashion, unless the search operator is used to do look-ahead (i.e., planning). Here the main strength of IndiGolog is that it can be used to generate plans that are epistemically adequate in the sense of section 3.6.2.

In the context of ALPs we have discussed the differences between online and offline execution in section 3.7. In particular, the online execution of ALPs has been characterized by restricting substitutions in the proof calculus to be non-disjunctive. We have likewise characterized epistemically adequate planning via ALPs by restricting disjunctive substitutions to sense fluents in section 3.6.2. Next we observe that the distinction between online and offline execution is meta-logical. It is hence not hard

[1] On a sidenote, let us again shortly point out that this transition semantics again is readily generalized by using the UAC for the underlying action theory.

to imagine how the ALP proof calculus can be adapted to interleaved online and offline execution: We can, e.g., introduce two reserved constants `online` and `offline` into the language to switch between the respective modes of using the proof calculus.

Arguably, the technical apparatus underlying this ALP approach to interleaving online and offline reasoning is simpler than that of IndiGolog — the definitive treatment of the latter can be found in Sebastian Sardina's PhD thesis [Sardina, 2005].

6.1.4 Golog Interpreter

In this section we first turn to the question whether the basic Golog interpreter can be used for an implementation of the ALP framework.

This original interpreter for basic Golog [Levesque et al., 1997] is based on the following ideas: First, regress the situation formula to be proved to the initial situation S_0. Then evaluate the regressed formula against a representation of the initial state by a set of Prolog facts. It is intuitively clear that this interpreter can be used for implementing yet another fragment of the ALP framework.

There only is the minor nuisance that it is not entirely clear theoretically which initial situations can be represented in this fragment:[2] The definition of a "closed initial database" (cf. p. 97 of [Reiter, 2001a]) does not rule out the formula $(\forall x)F(x, S_0) \equiv x = A \vee F(x, S_0)$. On the other hand, in the remarks following the definition it is claimed that with a closed initial database "... one cannot say that .... $F(A, S_0)$ is all that is known to be true of F.". Even though this issue is of little practical relevance (practically one just specifies the initial database by Prolog facts), it would be nice to find out exactly which class of initial situations the basic Golog interpreter can handle.

6.2 Flux

In this section we discuss the relation of the Flux language [Thielscher, 2005a, Thielscher, 2005d], and the ALP framework. In particular, we discuss

- the semantics of Flux and ALPs;
- the use of Flux as reasoner for action domains within the ALP framework;
- the advanced features of Flux not yet present in ALPs; and
- the relation of Flux to our own implementation work.

[2] The following observation was made by Stephan Schiffel.

6.2.1 Semantics of Flux

Flux programs are full constraint logic programs together with an implementation of a fragment of the Fluent Calculus via constraint handling rules (CHR) [Frühwirth, 1998]. Thus they may contain constructs like the cut and negation-as-failure, and the semantics of Flux programs accordingly has been given in terms of computation trees for constraint logic programs [Thielscher, 2005a].

Flux is closely tied to an earlier version of the Fluent Calculus that contains an explicit axiomatization of a state [Thielscher, 2005d]: Flux programs contain state terms, which do not exist in other action calculi. Let us illustrate this state based version of the Fluent Calculus: For example, the formula $\mathrm{Holds}(F, \mathrm{State}(s))$ is expanded to

$$\exists z \mathrm{State}(s) = z \wedge \exists z' z = F \circ z',$$

where z, z' are variables of sort STATE, and $\circ$ is the state forming operator. Due to the presence of state terms it is not immediate to give a semantics for Flux programs using a different action calculus than Fluent Calculus.

But the basic idea of Flux and ALPs is very similar: Use logic programs to define the strategic behavior of an agent in a dynamic domain. The ALP framework can be seen as a variant of the Flux language that

- admits a declarative, logical semantics;
- fully separates between action domain and strategy — there are no state update commands in the strategy program; and
- can be instantiated by other action calculi than the Fluent Calculus.

6.2.2 Flux as Action Domain Reasoner for ALPs

We can make good use of the Flux kernel for implementing yet another fragment of the ALP framework by making the following observations.

The presence of state terms in the language does not pose any real problems: In [Drescher and Thielscher, 2007] it has been proved that every FO sentence can equivalently be represented by a Fluent Calculus state formula (with state terms). This observation can be adapted to the representation of UAC state formulas as Fluent Calculus state formulas. Or we can use a result from [Schiffel and Thielscher, 2006] that Fluent Calculus domains (again with state terms) can equivalently be represented in the Situation Calculus (and vice versa). Either way, it is clear that the Fluent Calculus domains (with state terms) can be used to faithfully represent UAC action domains.

The biggest problem is that the `holds` atom in Flux is implemented such that it evaluates to true whenever it does not lead to a contradiction. But for ALPs we

want that `?` atoms evaluate to true only if they are entailed. For this we can use something similar to the following:[3]

```
?(F) :- \+ not_holds(F).
?(neg F) :- \+ holds(F).
```

This is the same technique that is used in Flux for knowledge based programming [Thielscher, 2005d].

Thus the Flux kernel can be used in an ALP implementation for representing actions, their effects, and state knowledge. Observe that the reasoning about state knowledge in Flux is sound, but not complete — for the details of the nature of incomplete reasoning in Flux the reader is referred to [Thielscher, 2005d] and [Thielscher, 2002].

Let us now discuss the expressivity of state properties that can be used in Flux for action preconditions, `?(Phi)` queries, and for representing the agent's state knowledge:

Definition 6.1 (Flux-expressible state property)**.** *A state property is expressible in Flux if it is of the form*

$$(\exists \vec{x}) \bigwedge_i \varphi_i \wedge \bigwedge_j \Psi_j(\vec{x})$$

where each φ_i is a Holds atom with variables among $\vec{x}$ and each $\Psi_j(\vec{x})$ (in the original version of the Flux language) is one of the following:

- *$\neg\varphi$, where φ is a Holds atom with variables among $\vec{x}$; or*
- *$(\forall \vec{y})\neg\varphi$, where φ is a Holds atom with variables among $\vec{y}$; or*
- *$\bigvee_k \varphi_k$, where each φ_k is a Holds atom, or an equality with variables among $\vec{x}$; or*
- *an arithmetic finite domain constraint with variables among $\vec{x}$.*

In [Thielscher, 2005c] the Flux language has been extended to cover so-called universal and implicational constraints. These allow that $\Psi_j(\vec{x})$ can also be one of the following:

- *$\varphi_1 \supset \bigvee_l \varphi_l$, where each φ_i is a Holds atom, or an (in-)equality with variables among $\vec{x}$; or*
- *$(\forall \vec{y})\varphi_1 \supset \varphi_2$, where φ_1 is a finite domain constraint with variables among $\vec{y}$ and φ_2 is a literal with variables $\vec{y}$.*

[3] Note that in Flux we can only query single fluents.

Flux deals with arithmetic by using the finite domain constraint libraries of ECLiPSe Prolog. With regard to both expressivity and practical efficiency of the language this constitutes a big advantage. Let us hence point out again that our ALP framework already features the same expressive means. In particular, this means we may use the same finite domain constraint libraries that Flux is using to supplement our ABox update and ALPprolog implementations — or any other future implementation of a fragment of the ALP framework.

Next observe that for the Flux language with arbitrary terms as fluents reasoning about a single state is tractable — disregarding the arithmetic constraints. This has first been made formally precise by Hannes Straß.[4] This is quite a striking contrast with ALPprolog (NP-complete), and ABox update (PSpace- or NExpTime-complete).

6.2.3 Advanced Features of Flux

Let us comment briefly on the non-logical features available in Flux: We have already discussed how the ALP framework can be extended by incorporating negation as finite failure, and by subsequently appealing to the soundness and (restricted) completeness results for the CLP($\mathcal{X}$)-scheme — cf. section 3.3.3. The other non-logical feature of Flux, namely the cut, is very useful for programming agent strategies that are meant for online execution, where backtracking simply is not possible. But ALPs already feature a principled distinction between online and offline reasoning. Making the cut available in ALPs will preclude a completeness result. If combined with negation as failure it will even preclude a soundness result [J.W. Lloyd, 1987].

The formalize notion of knowledge in Fluent Calculus (and also Flux) allows to make an interesting distinction: We can distinguish an agent not knowing something from an agent that knows that this something is not the case. In the plain ALP framework this distinction is not possible: We read `neg ?(Phi)` as `?(neg Phi)` (cf. section 3.3.3 on negation as finite failure).

Flux has also been applied to model action domains that are not modular, i.e. with implicit preconditions (so-called qualifications), and implicit effects (so-called ramifications) [Thielscher, 2005d]. The former have been modeled via Default Logic, whereas the latter appeal to the transitive closure of so-called causal relationships. In both cases second order notions are involved — something we so far try to avoid in the ALP framework. However, if the action domains used in practical implementations are not modular, then both these issues have to be addressed. Incorporating default reasoning about action effects in planning problems formulated in the UAC is the subject of ongoing work [Straß and Thielscher, 2009] — once this has been achieved ALPs at least theoretically can also be used for planning atop of action

[4] Personal Communication.

domains with implicit effects.

Finally, Flux has also been used to infer epistemically adequate conditional plans [Thielscher, 2001, Thielscher, 2005d]. This, however, was done using an extension of the underlying action formalism — the notion of actions was generalized to complex actions as terms of the language. Arguably, here the ALP formulation is more elegant — but Flux has the indisputable advantage of an existing implementation.

6.3 Domain-independent Planners

In this section we discuss the relation of the ALP framework to the existing work on domain-independent planning systems. Let us first recall the generic formulation of a planning problem as an ALP (cf. example 3.1):

```
strategy :- ?(goal).
strategy :- do(A), strategy.
```

This program uses the simplest possible strategy to find a plan — no domain dependent heuristics have been included. Of course, we can solve such a planning problem using the ALP proof calculi: The soundness and completeness results assure us that we will eventually find the solution (if it exists). However, for this generic ALP we may be able to find the solution much faster if we use a planner, since over the last decades much research effort went into the development of efficient, domain-independent planners. Of course, for this the planning language has to be expressible in the UAC — and vice versa, the action domain has to be expressible in the planning language.

Research in specialized planning languages originates with the STRIPS language [Fikes and Nilsson, 1971]. Over the years this basic language for specifying planning problems has seen numerous expressive extensions: first to the language ADL [Pednault, 1989] and then to PDDL [Ghallab et al., 1998], the ever growing language that underlies the annual planning competitions.

Traditionally the semantics of planning languages is given in terms of state transitions. There also is a parallel line of research that seeks to provide a logical semantics for planning languages. Usually this is done by mapping the planning problems of a given planning language to an equivalent representation in a (fragment of a) logic-based action calculus. Now if the action theory D underlying the generic planning ALP is in (or, can be mapped to) the corresponding fragment then we can resort to dedicated planning software for the evaluation of the generic planning ALP.

Such complementary semantics have already been developed both for STRIPS [Lifschitz, 1986, Lin and Reiter, 1995] and the more expressive ADL [Pednault, 1994, Claßen and Lakemeyer, 2006, Thielscher, 2007]. Both STRIPS and ADL planning domains hence fall within the ALP framework, and hence the existing mature

implementations can be used. There are also recent works [Claßen et al., 2007b, Claßen et al., 2007a] that aim at successively covering all of the semantics of PDDL 2.1 [Fox and Long, 2003]. These works, however, deal with a planning language that features concurrent actions — this does not fit the requirement that action domains underlying ALPs should be sequential for planning completeness (cf. definition 3.8).

In [Drescher and Thielscher, 2008] we have given a Fluent Calculus semantics for the fragment of PDDL 3.0 consisting of basic ADL and plan constraints that have been introduced to PDDL 3.0 in [Gerevini and Long, 2006]. This fragment again lies within the range of the ALP framework. Only a minor tweak on the definition of a state property and macro-expansion is required: We want to express planning goals like, e.g., $(\exists s)\phi(s) \wedge \neg(\exists s')s' \leq s \wedge \psi(s')$ — stating that on our way to the goal $\phi(s)$ at no intermediate timepoint s' the state property $\psi(s')$ held.

But maybe we want to use action domains D that are beyond the range of today's planners together with the generic planning ALP. Then it may still be possible to use some of the ideas underlying the fastest planners to date: Modern planners for example use simplified approximations of the planning problem (e.g., by ignoring the negative effects of actions) to derive a heuristics for picking actions to solve the full planning problem. Adapting these ideas, and implementing them, so that they can be put to use in fully general UAC planning domains is also left as a nice research challenge for future work.

6.4 Future Action Domain Reasoners

In this section we give some pointers to recent theoretical advances that may result in future, practically efficient action domain reasoners.

6.4.1 Progression in Situation Calculus

To date the dominant reasoning method in the Situation Calculus is regression, not progression (for an informal introduction to pro- and regression cf. chapter **??**). The theoretical work on progression in the Situation Calculus originates with [Lin and Reiter, 1997], where it was conjectured that progression in general is not first order definable, but needs second order logic. This conjecture has only recently been confirmed [Vassos and Levesque, 2008].

But in the same thrust of research also important classes of action domains in the Situation Calculus have been identified for which progression indeed is first order definable. The point that empirically progression seems to behave better in implementations than regression, has also surfaced in this thesis — especially in chapter 4. Hence the recent theoretical advances on progression in the Situation Calculus are quite promising. A PhD-thesis entirely concerned with the theoretical

aspects of progression is that of Stavros Vassos [Vassos, 2009] — the most recent results wrt. progression can be found in [Liu and Lakemeyer, 2009].

6.4.2 Proper Knowledge Bases and Situation Calculus

Levesque has introduced the notion of evaluation-based reasoning with disjunctive knowledge. The aim of this work is to have open world knowledge bases that admit reasoning as efficient as classical, closed world relational databases [Levesque, 1998, Liu and Levesque, 2003] — such a knowledge base is called a proper KB.

In [Liu and Levesque, 2005, Liu and Lakemeyer, 2009] it has been shown that proper KBs can be progressed, i.e. we can use an update algorithm rather than the regression algorithm usually employed in Golog implementations. Since empirically progression is superior to regression, and reasoning with proper KBs moreover is tractable, an actual implementation of Golog or the ALP framework based on the respective ideas appears to be very promising.

7 Conclusion

Let us now conclude our work. We will first give a summary of what we have achieved. Then we will give some pointers to interesting future work and open problems.

7.1 Summary

In this thesis we have presented Action Logic Programs — that is, we have proposed to use classical first order Horn clauses to specify strategies for

- the online control of agents, and
- offline planning,

both in dynamic domains represented by an axiomatization in an action calculus. The ALP framework stands apart from existing agent control languages by its straightforward declarative semantics — the classical FO semantics — and by not being tied to a particular action calculus.

We have identified the notion of a query-complete action domain — these are exactly those domains that can be combined with Prolog in a straightforward manner. If a planning problem in a query-complete domain has a solution then this plan is unconditional.

We have also presented sound and complete proof calculi: The LP(D) proof calculus for action domains that are query-complete, and the CLP(D) proof calculus for action domains that are not. These proof calculi appeal to ideas from classical (constraint) logic programming. In particular, these proof calculi together with a generic planning ALP provide a neat characterization of

- conditional planning (full CLP(D) calculus), and
- conformant planning (LP(D) proof calculus on arbitrary action domains),

both for action domains specified using full first order logic. Previous logical characterizations of these planning notions using, e.g., classical propositional logic or quantified Boolean formulas were limited to the propositional case.

We have also provided a novel and particularly simple approach to model planning in the presence of sensing actions: We use domain constraints to express the possible sensing results together with their meanings. By imposing simple restrictions on the

proof calculi ALPs can also be used to generate plans that are epistemically adequate for agents equipped with sensors.

The ALP framework can also be used for interleaving the online execution of strategies with the offline generation of epistemically adequate plans.

For two fragments of the general framework we have developed implementations aimed at the online control of agents:

The first of these is ALPprolog. ALPprolog is based on a syntactic variant of propositional logic and a concise state representation using prime implicates. It exploits our modeling of sensing actions via domain constraints. ALPprolog has proven to be considerably more efficient then the existing implementations of the logic-based action programming languages Golog and Flux on essentially propositional action domains that are supported by all three languages.

The second implementation we presented is based on a fragment of the Fluent Calculus where we use DL ABoxes to represent state knowledge. In the absence of competitors this implementation is harder to evaluate. On the one hand our implementation of the theoretically exponential ABox update proved to be practically superior to the theoretically polynomial so-called projective ABox update — this was a pleasant surprise. On the other hand our implementation still has its limitations. The variant that uses $\mathcal{ALCO}^{@}$ as underlying DL suffers from problems with keeping the state representation small, and the variant based on $\mathcal{ALCO}^{+}$ suffers from the absence of a truly satisfactory reasoner.

7.2 Directions for Future Work

There are a number of interesting directions for future work:

Clearly it would be nice to have a fast reasoner for Boolean $\mathcal{ALCO}^{+}$ ABoxes in order to fully exploit the potential of ABox update. But the most important task on the implementation side is to develop support for conditional planning problems.

On the theoretical side the following generalization of the ALP framework poses some interesting research questions: How can we accommodate concurrent actions in the underlying action domains? Is this even feasible while retaining the logical semantics, or is a transition semantics like that of Golog really called for?

Bibliography

[Andrews, 1986] Andrews, P. B. (1986). *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Orlando, FL.

[Apt, 1996] Apt, K. R. (1996). *From logic programming to Prolog*. Prentice-Hall.

[Areces et al., 1999] Areces, Blackburn, and Marx (1999). A road-map on complexity for hybrid logics. In *CSL: 13th Workshop on Computer Science Logic*. LNCS, Springer-Verlag.

[Areces and de Rijke, 2001] Areces, C. and de Rijke, M. (2001). From Description Logics to Hybrid Logics, and Back. In *Advances in Modal Logic*.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

[Baader et al., 2005] Baader, F., Lutz, C., Milicic, M., Sattler, U., and Wolter, F. (2005). Integrating description logics and action formalisms: First results. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, Pittsburgh, Pennsylvania. AAAI Press.

[Baader and Peñaloza, 2008] Baader, F. and Peñaloza, R. (2008). Automata-Based Axiom Pinpointing. In *Proceedings of the 4th International Joint Conference on Automated Reasoning, (IJCAR 2008)*. Springer.

[Baral, 2003] Baral, C. (2003). *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Cambridge, England.

[Baumgartner et al., 1997] Baumgartner, P., Furbach, U., and Stolzenburg, F. (1997). Computing answers with model elimination. *Artificial Intelligence*, 90(1–2):135–176.

[Bibel, 1986] Bibel, W. (1986). A deductive solution for plan generation. *New Generation Computing*, 4:115–132.

[Bibel, 1998] Bibel, W. (1998). Let's plan it deductively! *Artificial Intelligence*, 103(1–2):183–208.

[Blackburn et al., 2001] Blackburn, P., de Rijke, M., and Venema, Y. (2001). *Modal Logic*. Cambridge University Press.

[Bong, 2007] Bong, Y. (2007). Description Logic ABox Updates Revisited. Master thesis, TU Dresden, Germany.

[Bordini et al., 2007] Bordini, R., Hübner, J., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley.

[Borgida, 1996] Borgida, A. (1996). On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1-2):353–367.

[Brewka et al., 2008] Brewka, G., Niemelä, I., and Truszczýnski, M. (2008). Nonmonotonic reasoning. In van Harmelen, F., Lifschitz, V., and Porter, B., editors, *Handbook of Knowledge Representation*, chapter 6, pages 239–284. Elsevier Science, Amsterdam.

[Brouwer, 1907] Brouwer, L. E. J. (1907). *Over de Grondslagen der Wiskunde*. PhD thesis, University of Amsterdam. English translation in *L.E.J. Brouwer: Collected Works 1: Philosophy and Foundations of Mathematics* (A. Heyting, Editor), Elsevier, Amsterdam and New York, 1975.

[Burhans and Shapiro, 2007] Burhans, D. T. and Shapiro, S. C. (2007). Defining Answer Classes Using Resolution Refutation. *Journal of Applied Logic*, 5(1):70–91.

[Castilho et al., 1999] Castilho, M. A., Gasquet, O., and Herzig, A. (1999). Formalizing action and change in modal logic I: The frame problem. *Journal of Logic and Computation*, 9(5):701–735.

[Chang and Keisler, 1990] Chang, C. C. and Keisler, H. J. (1990). *Model Theory*. North Holland.

[Clark, 1978] Clark, K. L. (1978). Negation as Failure. In Gallaire, H. and Minker, J., editors, *Logic and Data Bases*, pages 293–322. Plenum Press.

[Claßen et al., 2007a] Claßen, J., Eyerich, P., Lakemeyer, G., and Nebel, B. (2007a). Towards an integration of golog and planning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 07)*, Hyderabad, India.

[Claßen et al., 2007b] Claßen, J., Hu, Y., and Lakemeyer, G. (2007b). A situation-calculus semantics for an expressive fragment of PDDL. In *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI 2007)*, Menlo Park, CA.

[Claßen and Lakemeyer, 2006] Claßen, J. and Lakemeyer, G. (2006). A semantics for ADL as progression in the situation calculus. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR06)*, Lake District, UK.

[Clocksin and Mellish, 1987] Clocksin, W. F. and Mellish, C. (1987). *Programming in Prolog, 3rd Edition*. Springer.

[Davis, 1993] Davis, M. (1993). First order logic. In Gabbay, D., Hogger, C. J., and Robinson, J. A., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, chapter 2, pages 31–65. Oxford University Press.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A Machine Program for Theorem-proving. *Communications of the ACM*.

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A Computing Procedure for Quantification Theory. *Journal of the ACM*.

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*. Springer.

[Doherty et al., 1998] Doherty, P., Lukaszewicz, W., and Madalinska-Bugaj, E. (1998). The pma and relativizing minimal change for action update. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR 98)*, Trento, Italy. AAAI Press.

[Drescher et al., 2009] Drescher, C., Liu, H., Baader, F., Guhlemann, S., Petersohn, U., Steinke, P., and Thielscher, M. (2009). Putting abox updates into action. In *Proceedings of the Seventh International Symposion on Frontiers of Combining Systems (FroCoS 2009)*, Trento, Italy.

[Drescher and Thielscher, 2007] Drescher, C. and Thielscher, M. (2007). Integrating action calculi and description logics. In *Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI 2007)*, Osnabrück, Germany.

[Drescher and Thielscher, 2008] Drescher, C. and Thielscher, M. (2008). A fluent calculus semantics for ADL with plan constraints. In *Proceedings of the 11th European Conference on Logics in Artificial Intelligence„ JELIA08 28 - October 1, 2008. Proceedings*, Dresden, Germany. Springer.

[Dummett, 1977] Dummett, M. (1977). *Elements of Intuitionism*. Oxford University Press, Oxford.

[Ebbinghaus and Flum, 1995] Ebbinghaus, H.-D. and Flum, J. (1995). *Finite Model Theory*. Perspectives in Mathematical Logic. Springer.

[Ebbinghaus et al., 1994] Ebbinghaus, H.-D., Flum, J., and Thomas, W. (1994). *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, Berlin, 2nd edition. (1st ed., 1984).

[ECLiPSe Implementors Group, 2009] ECLiPSe Implementors Group (2009). *ECLiPSe User Manual*. `http://www.eclipse-clp.org`.

[Een and Sörensson, 2003] Een, N. and Sörensson, N. (2003). An Extensible SAT-solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*.

[Enderton, 1972] Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. Academic Press.

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

[Fox and Long, 2003] Fox, M. and Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

[Frege, 1879] Frege, G. (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle. English translation in *From Frege to Gödel, a Source Book in Mathematical Logic* (J. van Heijenoort, Editor), Harvard University Press, Cambridge, 1967, 1–82.

[Frühwirth, 1998] Frühwirth, T. (1998). Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3).

[Frühwirth and Abdennadher, 2003] Frühwirth, T. and Abdennadher, S. (2003). *Essentials of Constraint Programming*. Springer.

[Gabbay et al., 1998] Gabbay, D., Hogger, C., and Robinson, J. A., editors (1992-1998). *Handbook of Logic in Artificial Intelligence and Logic Programming, Volumes 1-5*. Oxford University Press, Oxford.

[Gelfond and Lifschitz, 1992] Gelfond, M. and Lifschitz, V. (1992). Describing action and change by logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (ICLP-92)*, Washington, DC.

[Genesereth et al., 2005] Genesereth, M. R., Love, N., and Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62–72.

[Gerevini and Long, 2006] Gerevini, A. and Long, D. (2006). Preferences and soft constraints in pddl3. In *Proceedings of the ICAPS-2006 Workshop on Preferences and Soft Constraints in Planning*, Lake District of the UK.

[Ghallab et al., 1998] Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL—the planning domain definition language.

[Giacomo et al., 2000] Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2000). Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169.

[Girard, 1987] Girard, J. (1987). Linear logic. *Journal of Theoretical Computer Science*, 50:1–102.

[Girard, 2001] Girard, J.-Y. (2001). Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506.

[Gödel, 1930] Gödel, K. (1930). *Über die Vollständigkeit des Logikkalküls*. PhD thesis, University of Vienna.

[Götzmann, 2009] Götzmann, D. (2009). Spartacus: A tableau prover for hybrid logic. Master thesis, Saarland University, Germany.

[Green, 1969] Green, C. (1969). Theorem proving by resolution as a basis for question-answering systems. *Machine Intelligence*, 4:183–205.

[Herzig and Varzinczak, 2007] Herzig, A. and Varzinczak, I. (2007). Metatheory of actions: Beyond consistency. *Artificial Intelligence*, 171(16–17):951–984.

[Hill, 1974] Hill, R. (1974). LUSH resolution and its completeness. Technical Report DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh.

[Hodges, 1997] Hodges, W. (1997). *A Shorter Model Theory*. Cambridge University Press.

[Hölldobler and Schneeberger, 1990] Hölldobler, S. and Schneeberger, J. (1990). A new deductive approach to planning. *New Generation Computing*, 8:225–244.

[Horrocks and Voronkov, 2006] Horrocks, I. and Voronkov, A. (2006). Reasoning support for expressive ontology languages using a theorem prover. In *Proceedings of the 4th International Symposion on the Foundations of Information and Knowledge Systems (FoIKS 2006)*, Lecture Notes in Computer Science, pages 201–218, Budapest, Hungary. Springer.

[Jaffar and Lassez, 1987] Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, Munich.

[Jaffar et al., 1998] Jaffar, J., Maher, M. J., Marriott, K., and Stuckey, P. J. (1998). The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46.

[J.W. Lloyd, 1987] J.W. Lloyd (1987). *Foundations of Logic Programming*. Springer.

[Kazakov and Motik, 2008] Kazakov, Y. and Motik, B. (2008). A Resolution-Based Decision Procedure for SHOIQ. *Journal of Automated Reasoning*.

[Kowalski and Sergot, 1986] Kowalski, R. A. and Sergot, M. J. (1986). A Logic-Based Calculus of Events. *New Generation Computing*, 4:67–95.

[Lakemeyer and Levesque, 2004] Lakemeyer, G. and Levesque, H. J. (2004). Situations, si! situation terms, no! In *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR 04)*, Whistler, Canada. AAAI Press.

[Lakemeyer and Levesque, 2005] Lakemeyer, G. and Levesque, H. J. (2005). Semantics for a useful fragment of the situation calculus. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 05)*, Edinburgh, Scotland.

[Leivant, 1994] Leivant, D. (1994). Higher order logic. In Gabbay, D. M., Hogger, C. J., and Robinson, J. A., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1: Deduction Methodologies, pages 229–322. Oxford University Press.

[Levesque et al., 1997] Levesque, H., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. (1997). GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83.

[Levesque, 1996] Levesque, H. J. (1996). What is planning in the presence of sensing? In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*, pages 1139–1146, Portland, Oregon, USA.

[Levesque, 1998] Levesque, H. J. (1998). A completeness result for reasoning with incomplete first-order knowledge bases. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR 98)*, San Francisco, California.

[Lifschitz, 1986] Lifschitz, V. (1986). On the semantics of STRIPS. In *Reasoning about actions and plans: Proceeding of 1986 workshop*, Temberline, Oregon.

[Lifschitz, 1994] Lifschitz, V. (1994). Circumscription. In Gabbay, D., Hogger, C., and Robinson, A., editors, *Handbook of Logic in AI and Logic Programming*, volume 3, pages 298–352. Oxford University Press.

[Lifschitz et al., 2007] Lifschitz, V., Porter, B., and van Harmelen, F., editors (2007). *Handbook of Knowledge Representation*. Elsevier.

[Lin and Reiter, 1995] Lin, F. and Reiter, R. (1995). How to Progress a Database II: The STRIPS Connection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 95)*, Montreal, Canada. Morgan Kaufmann.

[Lin and Reiter, 1997] Lin, F. and Reiter, R. (1997). How to progress a database. *Artificial Intelligence*, 92(1–2):131–167.

[Liu, 2009] Liu, H. (2009). *Updating Description Logic ABoxes*. PhD thesis, Dresden University of Technology, Germany.

[Liu et al., 2006] Liu, H., Lutz, C., Milicic, M., and Wolter, F. (2006). Updating description logic ABoxes. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR 06)*, Lake District of the UK.

[Liu and Lakemeyer, 2009] Liu, Y. and Lakemeyer, G. (2009). On first-order definability and computability of progression for local-effect actions and beyond. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI 09)*, Pasadena, California, USA.

[Liu et al., 2004] Liu, Y., Lakemeyer, G., and Levesque, H. J. (2004). A logic of limited belief for reasoning with disjunctive information. In *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR 04)*. AAAI Press, Menlo Park, California.

[Liu and Levesque, 2003] Liu, Y. and Levesque, H. J. (2003). A tractability result for reasoning with incomplete first-order knowledge bases. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 03)*, Acapulco, Mexico.

[Liu and Levesque, 2005] Liu, Y. and Levesque, H. J. (2005). Tractable reasoning with incomplete first-order knowledge in dynamic systems with context-dependent actions. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 05)*, Edinburgh, Scotland, UK.

[Love et al., 2008] Love, N., Hinrichs, T., Haley, D., Schkufza, E., and Genesereth, M. (2008). General game playing: Game description language specification. Technical report, Stanford University.

[Manna and Waldinger, 1987] Manna, Z. and Waldinger, R. J. (1987). How to clear a block: A theory of plans. *Journal of Automated Reasoning*, 3(4):343–377.

[Mascardi et al., 2005] Mascardi, V., Demergasso, D., and Ancona, D. (2005). Languages for programming BDI-style agents: an overview. In *6th Workshop on From Objects to Agents*, Camerino, Italy.

[McCarthy, 1963] McCarthy, J. (1963). *Situations and Actions and Causal Laws.* Stanford Artificial Intelligence Project, Memo 2, Stanford University, CA.

[McCarthy, 1980] McCarthy, J. (1980). Circumscription—A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13(1–2):27–39.

[McCarthy and Hayes, 1969] McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press.

[McCune, 2003] McCune, W. (2003). OTTER 3.3 Manual. *Computing Research Repository.*

[Milicic, 2008] Milicic, M. (2008). *Action, Time, and Space in Description Logics.* PhD thesis, Dresden University of Technology, Germany.

[Miller and Shanahan, 2002] Miller, R. and Shanahan, M. (2002). Some alternative formulations of the event calculus. In Kakas, A. C. and Sadri, F., editors, *Computational Logic. Logic Programming and Beyond*, pages 452–490. Springer.

[Mueller, 2006] Mueller, E. T. (2006). *Commonsense Reasoning.* Morgan Kaufmann.

[Nieuwenhuis et al., 2007] Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., and Rubio, A. (2007). Challenges in Satisfiability Modulo Theories. In *18th International Conference on Term Rewriting and Applications.* Springer.

[Nonnengart and Weidenbach, 2001] Nonnengart, A. and Weidenbach, C. (2001). Computing small clause normal forms. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science B.V.

[O'Keefe, 1990] O'Keefe, R. A. (1990). *The Craft of Prolog.* MIT Press.

[Pednault, 1989] Pednault, E. P. D. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR 89)*, San Mateo, California.

[Pednault, 1994] Pednault, E. P. D. (1994). ADL and the state-transition model of action. *Journal of Logic and Computation*, 4(5):467–512.

[Pirri and Reiter, 1999] Pirri, F. and Reiter, R. (1999). Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–361.

[Pratt-Hartmann, 2005] Pratt-Hartmann, I. (2005). Complexity of the two-variable fragment with counting quantifiers. *Journal of Logic, Language, and Information*, 14(3):369–395.

[Reiter, 1991] Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 359–380. Academic Press.

[Reiter, 2001a] Reiter, R. (2001a). *Knowledge in Action*. MIT Press.

[Reiter, 2001b] Reiter, R. (2001b). *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, MA.

[Restall, 2000] Restall, G. (2000). *An Introduction to Substructural Logics*. Routledge.

[Robinson, 2000] Robinson, J. A. (2000). Computational logic: Memories of the past and challenges for the future. In *Proceedings of the First International Conference on Computational Logic (CL 2000)*, London, UK. Springer.

[Russell and Norvig, 2003] Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: a modern approach*. Prentice Hall, Upper Saddle River, N.J., 2nd international edition edition.

[Sardina, 2005] Sardina, S. (2005). *Deliberation in Agent Programming Languages*. PhD thesis, Department of Computer Science.

[Sardina et al., 2004] Sardina, S., De Giacomo, G., Lespérance, Y., and Levesque, H. J. (2004). On the semantics of deliberation in IndiGolog – From theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):259–299.

[Sardiña et al., 2004] Sardiña, S., Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2004). On ability to autonomously execute agent programs with sensing. In *Proceedings of the 4th International Workshop on Cognitive Robotics (CoRobo-04)*, Valencia,Spain. IEEE Computer Society.

[Savelli, 2006] Savelli, F. (2006). Existential assertions and quantum levels on the tree of the situation calculus. *Artificial Intelligence*, 170(2):643–652.

[Schaerf, 1994] Schaerf, A. (1994). Reasoning with individuals in concept languages. *Data and Knowledge Engineering*, 13:141–176.

[Scherl and Levesque, 2003] Scherl, R. B. and Levesque, H. (2003). Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1–2):1–39.

[Schiffel and Thielscher, 2006] Schiffel, S. and Thielscher, M. (2006). Reconciling situation calculus and fluent calculus. In *Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI 2006)*, Boston, MA. AAAI Press.

[Schiffel and Thielscher, 2007] Schiffel, S. and Thielscher, M. (2007). Fluxplayer: A successful general game player. In *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI 2007)*, Menlo Park, CA. AAAI Press.

[Schlobach, 2003] Schlobach, S. (2003). Non-Standard Reasoning Services for the Debugging of Description Logic Terminologies. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, (IJCAI-03)*. Morgan Kaufmann.

[Schmidt and Tishkovsky, 2007] Schmidt, R. A. and Tishkovsky, D. (2007). Using tableau to decide expressive description logics with role negation. In *Proceedings of the 6th International Semantic Web Conference, ISWC 2007*. Springer.

[Shanahan, 1995] Shanahan, M. (1995). A circumscriptive calculus of events. *Artificial Intelligence*, 77(2):251–284.

[Shanahan, 1997] Shanahan, M. (1997). *Solving the Frame Problem*. MIT Press, Cambridge, Massachusetts.

[Shanahan, 2000] Shanahan, M. (2000). An abductive event calculus planner. *Journal of Logic Programming*, 44(1-3):207–240.

[Shanahan, 2002] Shanahan, M. (2002). A logical account of perception incorporating feedback and expectation. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR 02)*. Morgan Kaufmann, San Francisco, California.

[Shanahan and Randell, 2004] Shanahan, M. and Randell, D. A. (2004). A logic-based formulation of active visual perception. In *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR 04)*.

[Shanahan and Witkowski, 2000] Shanahan, M. and Witkowski, M. (2000). High-level robot control through logic. In *Proceedings of the International Workshop on Agent Theories, Architectures and Languages (ATAL)*, Boston, MA.

[Shapiro, 1991] Shapiro, S. (1991). *Foundations without Foundationalism: A case for second-order logic*. Oxford University Press.

[Shoenfield, 1967] Shoenfield, J. R. (1967). *Mathematical Logic*. Addison-Wesley, Reading, MA.

[Sirin et al., 2007] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*.

[Spark Implementors Group, 2009] Spark Implementors Group (2009). *Spark Reference Manual*. `http://www.ai.sri.com/~spark`.

[Stärk, 1990] Stärk, R. F. (1990). A direct proof for the completeness of SLD-resolution. In *Third Workshop on Computer Science Logic*.

[Straß and Thielscher, 2009] Straß, H. and Thielscher, M. (2009). Simple default reasoning in theories of action. In *Proceedings of the 22nd Australasian Joint Conference on Artificial Intelligence(AI09)*, Melbourne, Australia.

[Thielscher, 1999a] Thielscher, M. (1999a). From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299.

[Thielscher, 1999b] Thielscher, M. (1999b). From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299.

[Thielscher, 1999c] Thielscher, M. (1999c). From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299.

[Thielscher, 2000] Thielscher, M. (2000). Representing the knowledge of a robot. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 00)*, Breckenridge, CO. AAAI Press.

[Thielscher, 2001] Thielscher, M. (2001). Inferring implicit state knowledge and plans with sensing actions. In Baader, F., Brewka, G., and Eiter, T., editors, *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, volume 2174 of *LNAI*, pages 366–380, Vienna, Austria. Springer.

[Thielscher, 2002] Thielscher, M. (2002). Reasoning about actions with CHRs and finite domain constraints. In Stuckey, P., editor, *Proceedings of the International Conference on Logic Programming (ICLP)*, volume 2401 of *LNCS*, pages 70–84, Copenhagen, Danmark. Springer.

[Thielscher, 2004] Thielscher, M. (2004). Logic-based agents and the frame problem: A case for progression. In Hendricks, V., editor, *First-Order Logic Revisited: Proceedings of the Conference 75 Years of First Order Logic (FOL75)*, pages 323–336, Berlin, Germany. Logos.

[Thielscher, 2005a] Thielscher, M. (2005a). FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4–5):533–565.

[Thielscher, 2005b] Thielscher, M. (2005b). A FLUX agent for the Wumpus World. In Morgenstern, L. and Pagnucco, M., editors, *Proceedings of the Workshop on Nonmonotonic Reasoning, Action and Change at IJCAI*, pages 104–108, Edinburgh, UK.

[Thielscher, 2005c] Thielscher, M. (2005c). Handling implicational and universal quantification constraints in flux. In *Proceedings of the International Conference on Principle and Practice of Constraint Programming (CP)*, Sitges, Spain. Springer.

[Thielscher, 2005d] Thielscher, M. (2005d). *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Kluwer.

[Thielscher, 2007] Thielscher, M. (2007). A Unifying Action Calculus. *Artificial Intelligence (submitted)*. `http://www.fluxagent.org/publications.htm`.

[Tobies, 2001] Tobies, S. (2001). *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH-Aachen, Germany.

[Tsarkov and Horrocks, 2003] Tsarkov, D. and Horrocks, I. (2003). DL reasoner vs. first-order prover. In Calvanese, D., Giacomo, G. D., and Franconi, E., editors, *Proceedings of the 2003 International Workshop on Description Logics (DL 2003)*, Rome, Italy. CEUR-WS.org.

[van Dalen, 1994] van Dalen, D. (1994). *Logic and Structure*. Universitext. Springer-Verlag, Berlin, 3rd, augmented edition. (1st ed., 1980; 2nd ed., 1983).

[van Emden and Kowalski, 1976] van Emden, M. H. and Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742.

[Vassos, 2009] Vassos, S. (2009). *A Reasoning Module for Long-Lived Cognitive Agents*. PhD thesis, University of Toronto, Toronto, Canada.

[Vassos and Levesque, 2008] Vassos, S. and Levesque, H. J. (2008). On the progression of situation calculus basic action theories: Resolving a 10-year-old conjecture. In *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI 2008)*, Chicago, Illinois, USA. AAAI Press.

[Volz, 2007] Volz, R. (2007). *Web ontology reasoning with logic databases*. PhD thesis, Universität Karlsruhe.

[Waldinger, 2007] Waldinger, R. J. (2007). Whatever happened to deductive question answering? In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, (LPAR 07), Yerevan, Armenia*. Springer.

[Winslett, 1988] Winslett, M. (1988). Reasoning about Action Using a Possible Models Approach. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI)*. AAAI Press.

[Winslett, 1990] Winslett, M. (1990). *Updating Logical Databases*. Cambridge University Press.

List of own Publications

[Drescher et al., 2009a] Drescher, C., Liu, H., Baader, F., Guhlemann, S., Petersohn, U., Steinke, P., and Thielscher, M. (2009a). Putting abox updates into action. In *Proceedings of the Seventh International Symposion on Frontiers of Combining Systems (FroCoS 2009)*, Trento, Italy.

[Drescher et al., 2009b] Drescher, C., Liu, H., Baader, F., Steinke, P., and Thielscher, M. (2009b). Putting abox updates into action. In *Proceedings of the 8th IJCAI International Workshop on Nonmonotonic Reasoning, Action and Change (NRAC-09)*, Pasadena, California, US.

[Drescher et al., 2009c] Drescher, C., Schiffel, S., and Thielscher, M. (2009c). A declarative agent programming language based on action theories. In *Proceedings of the Seventh International Symposion on Frontiers of Combining Systems (FroCoS 2009)*, Trento, Italy.

[Drescher and Thielscher, 2007a] Drescher, C. and Thielscher, M. (2007a). Integrating action calculi and description logics. In *Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI 2007)*, Osnabrück, Germany.

[Drescher and Thielscher, 2007b] Drescher, C. and Thielscher, M. (2007b). Reasoning about actions with description logics. In *Proceedings of the 7th IJCAI International Workshop on Nonmonotonic Reasoning, Action and Change (NRAC-07)*, Hyderabad, India.

[Drescher and Thielscher, 2008] Drescher, C. and Thielscher, M. (2008). A fluent calculus semantics for ADL with plan constraints. In *Proceedings of the 11th European Conference on Logics in Artificial Intelligence„ JELIA08*, Dresden, Germany. Springer.