Linköping Studies in Science and Technology

Thesis No. 1329

Deductive Planning and Composite Actions in Temporal Action Logic

by

Martin Magnusson

Linköping University
INSTITUTE OF TECHNOLOGY

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for degree of Licentiate of Engineering

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2007

Deductive Planning and Composite Actions in Temporal Action Logic

by

Martin Magnusson

September 2007
ISBN 978–91–85895–93–9
Linköping Studies in Science and Technology
Thesis No. 1329
ISSN 0280–7971
LiU–Tek–Lic–2007:38

ABSTRACT

Temporal Action Logic is a well established logical formalism for reasoning about action and change that has long been used as a formal specification language. Its first-order characterization and explicit time representation makes it a suitable target for automated theorem proving and the application of temporal constraint solvers. We introduce a translation from a subset of Temporal Action Logic to constraint logic programs that takes advantage of these characteristics to make the logic applicable, not just as a formal specification language, but in solving practical reasoning problems. Extensions are introduced that enable the generation of action sequences, thus paving the road for interesting applications in deductive planning. The use of qualitative temporal constraints makes it possible to follow a least commitment strategy and construct partially ordered plans. Furthermore, the logical language and logic program translation is extended with the notion of composite actions that can be used to formulate and execute scripted plans with conditional actions, non-deterministic choices, and loops. The resulting planner and reasoner is integrated with a graphical user interface in our autonomous helicopter research system and applied to logistics problems. Solution plans are synthesized together with monitoring constraints that trigger the generation of recovery actions in cases of execution failures.

This work is supported in part by a grant from the Swedish national aeronautics research program NFFP04 S4203 and a Swedish research council grant 50364201.

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Acknowledgments

I would like to thank my advisor Patrick Doherty for his contagious enthusiasm for research, for entrusting me with great freedom in pursuing whatever topics that have interested me, and for paying me to enjoy my hobby. Two other great researchers that I have had the pleasure of collaborating with are Jonas Kvarnström and Andrzej Szalas. I am looking forward to more opportunities for joint work in the future.

My colleges at the department of Computer Science at Linköping University has made it a wonderful environment to work in. I sincerely thank Per Nyblom, David Landén, Gustav Nordh, Per-Olof Petersson, Fredrik Heintz, and Tommy Persson for being such good friends. Thank you Jenny Ljung, Anna Maria Uhlin, Madeleine Häger Dahlqvist, and Piotr Rudol for invaluable assistance.

The choice to pursue academic studies was made with the encouragement and support from Eva-Lena Lengquist with whom I have had the great fortune to share many happy moments. Steve Pavlina has unknowingly helped me tremendously towards my goal through his innumerable inspirational writings on how to improve any and every area of your life. My final note of gratitude goes to my parents and sister for your unconditional love.

Contents

Chapter 1

Introduction

Artificial intelligence is the challenge of creating a thinking machine. Although definitions of intelligence and thinking are plentiful, some capabilities seem absolutely necessary for any system proposed as either. Of central importance is the capability for rational reasoning, especially reasoning about what can be done to achieve ones goals. Work within the methodology of formal logic provides a comprehensive toolset for correct reasoning and is thus a natural choice for progress toward this end. However, the standard philosophical logic turns out to be inadequate, and artificial intelligence researchers are therefore creating new powerful logics that are applicable to human-like commonsense reasoning as well as more traditional logical reasoning.

Temporal Action Logic (TAL) [8] is one such logic developed at Linköping University. TAL has, since its inception more than ten years ago, proven to be a highly versatile and expressive formalism. Nevertheless, many important areas of research can be identified that provide excellent opportunities for improving the logic. In this thesis we present extensions to TAL that make it applicable to deductive planning, i.e. reasoning about what actions achieve a given goal, and we touch upon the topic of composite actions, i.e. complex actions built from simpler actions, conditionals and loops.

This and most other artificial intelligence research pursued at Linköping University is part of the Autonomous Unmanned Aerial Vehicle (UAV) Technologies Laboratory and its long-term basic research initiative [6] that focuses on the development of autonomous robots that display high-level cognitive behaviour with the primary experimental platforms being autonomous UAVs. Having such working systems in mind helps put emphasis on techniques that

are not purely of theoretical interest but also useful in practice. We want to make logic work.

1.1 The Case for Deductive Planning

By *deductive planning* we shall understand the direct synthesis of plans through a deductive proof process in a logic of action and change. In contrast most of the work in automated planning makes use of algorithms, formalisms, and programs specialized for the planning task. This kind of automated planning has seen great progress and has developed into a well-defined research topic with specialized conferences and a respectable body of work. In light of this fact, one might question the need to cast the planning process as deduction.

To answer this question we would like to appeal to the bigger picture. The artificial intelligence challenge requires an attempt at building a generally intelligent system that is applicable to *all* problems of *any* complexity. To be practically possible this would seem to require formalisms and technologies that exhibit two important properties. First, note that it is clearly impossible to foresee all possible problems requiring intelligence, in advance of the artificial agent being confronted by them, and to develop special purpose solutions to each and every one of them. The alternative is a striving for a property of *uniformity* that would allow most problems to be expressed in a similar way, and different or unforeseen problems to be attacked using the same techniques. This necessitates the second property of *expressivity*, to have the ability to encode all the different complexities of problems, and to be able to reason with them in the same framework. Deductive planning contributes uniformity since the planning process is cast as deduction in the same way as other types of reasoning such as prediction. It also supports expressivity since the logics used are often among the most expressive formalisms known that can be used to encode complex problems in a well understood way.

Before we get too carried away, however, let us admit that the present state of deductive planning still has a long way to go before fulfilling these ideals. The point we would like to make is that work in this direction is important and that deductive planning enjoys properties that could help progress towards the long term (as well as the short term) goal.

1.2 Contributions

One of the distinguishing characteristics of Temporal Action Logic is its use of explicit time structures whose meaning can be explained through seman-

tic attachment or through axiomatizations in classical logic. This feature can be taken advantage of by combining automated theorem proving with specialized temporal constraint propagation algorithms for dealing with temporal structures in TAL narratives. First-order automated theorem provers can be inefficient and difficult to use in practical applications, but logic programming, exemplified by Prolog, is a well-known alternative that has been successfully applied to many logical reasoning problems. Although logic programming technologies are based on theorem proving, they sacrifice some of the expressiveness of first-order logic to gain simplicity of use and, often, efficiency.

We present a translation from a subset of the full TAL formalism to logic programs that use Prolog's finite domain constraint solver and the Constraint Handling Rules [10] framework. The resulting programs take advantage of these constraint solving tools to make TAL directly applicable to practical planning problems in our autonomous Unmanned Aerial Vehicle (UAV) system. TAL has been used in previous work as a formal specification language for a very powerful forward-chaining planner called TALplanner [7]. A formally specified TAL goal narrative is input into the procedural planner, which in turn outputs a plan narrative whose formal semantics is also based on TAL. In this case, TAL is used as a formal specification tool and the plan generation mechanism operates outside the logic. In contrast, this thesis introduces additional extensions to TAL that enable deductive planning at the object level rather than meta-theoretically. In order to do this both action occurrences and sets of action occurrences (a form of plan or narrative) will be introduced as terms and sets in the object language, respectively. In this manner, one may generate partially ordered plans deductively through the execution of the constraint logic program that was translated from the TAL narrative.

Since the reasoning is deductive, the process works both ways. One may provide a complete plan as input and deduce its effects, specify only declarative goals that need to be satisfied, or explore the middle ground between these extremes. Sometimes, though, one may want to provide a complete plan for the robot to execute that can not be expressed as a sequence of simple actions. Many tasks involve choices that are conditional on some property of the world, or the repeated application of some sequence of actions until a condition has been satisfied. We extend the TAL language with macros that enable the encoding of such composite actions and extend the constraint logic programming translation so that they can be executed in the same logic programming framework.

This framework, named PARADOCS for Planning And Reasoning As DeductiOn with ConstraintS, has been integrated with our UAV system and applied to a logistics scenario. A UAV operator can set up a mission graphically

through a user interface, which also visualizes the resulting plan and its execution. Value persistency constraints in the plan are continually monitored, and trigger failure recovery when violated. Recovery is possible by providing the plan fragment that has yet to be executed as input to the deductive planner and reasoner. It will complete the partial plan with additional recovery actions as needed to make it valid again.

1.3 Publications

Parts of this thesis have previously been presented in the following publications:

[24] Martin Magnusson and Patrick Doherty. Deductive planning with temporal constraints using TAL. In Proceedings of the International Symposium on Practical Cognitive Agents and Robots (PCAR'06), 2006.

[25] Martin Magnusson and Patrick Doherty. Deductive planning with temporal constraints. In Logical Formalizations of Commonsense Reasoning: Papers from 2007 AAAI Spring Symposium, Technical Report SS-07-05, 2007.

1.4 Thesis Outline

The structure of this thesis is as follows. Chapter 2 provides an introduction to Temporal Action Logic sufficient to support understanding of the subsequent chapters. Chapter 3 introduces extensions that enable deductive planning with TAL and provides an example constructive proof of a simple planning goal from which a solution plan can be extracted. Chapter 4 details the compilation process that automates the planning process using constraint logic programming and temporal constraint formalisms. Chapter 5 specifies the semantics of composite actions and complements the compilation process so that they can be evaluated. Chapter 6 presents an integration of the developed methods with the UAV platform and demonstrates the application of the system to a logistics problem. Chapter 7 relates the work to some of its sources of inspiration. Chapter 8 sums up the results and discusses future directions of research. Finally, an appendix contains the Prolog clauses that were used in the logistics application.

Chapter 2

Temporal Action Logic

Temporal Action Logic has its roots in Sandewall's book *Features and Fluents* [32] published in 1994 but has since then been developed in a direction of its own by Doherty, Karlsson, Gustafsson, Kvarnström, and others (see e.g. [5, 17, 12, 19]). TAL differs from the well-known Situation Calculus and other logical formalisms for reasoning about action and change with its use of explicit time, occlusion, and a high-level language that provides an abstraction layer free from technical constructions introduced to deal with the frame and other problems.

This introduction to TAL is meant to be self contained, but by no means a complete feature list. A more detailed presentation of TAL is available elsewhere [8].

2.1 Basic Concepts

Most logics have an ontological base in properties and relations between objects. In TAL, properties and relations are modelled using *fluents* that assume *values*. Fluents are typed, so a given fluent only accepts arguments and only takes on values from the pre-specified *domains*. E.g., the location of a robot could be modelled using a boolean fluent *loc(robot,location)* that is associated with the value *true* or *false* for robot and location arguments from the domains *robot* and *location*. Alternatively one could use a *location*-valued fluent *loc(robot)* that has as value the object representing the location of the robot.

Fluents could be written as predicates in first-order logic. However, predicates can not have any values other than true or false and never change their values. In contrast, the value of the *loc* fluent can be a location that may change

over time as the robot moves around. One could get around this problem by introducing a timepoint and a value argument to the predicate. However, by representing fluents as terms instead of predicates one also gains the possibility of quantifying over them. A special predicate *Holds(timepoint,fluent,value)* is introduced to associate a value with a fluent term at a specific *timepoint*. Timepoints are explicit objects in the form of numbers or terms that refer to a time-line consisting of positive integers starting with zero.

Suppose we are interested in robotic UAVs and would like to model their ability to move to new locations. The effects of an action *fly(uav,location)* would be formally specified and that specification could later be used to reason about the consequences of occurrences of this action at specific timepoints. An *Occurs(timepoint,timepoint,action)* predicate is used for these purposes and represents an action occuring over the time interval between two timepoints.

Finally, the concept of *occlusion* is introduced as a very flexible solution to the frame and related problems. The basic idea is to make fluent values persist over time by minimizing their opportunities for change. A predicate *Occlude(timepoint,fluent)* represents the possibility of a fluent to change its value. Negated occlusion is then the property of a fluent retaining its value from the previous timepoint, axiomatized as:

$$\forall t\,[\neg \mathit{Occlude}(t+1, f) \rightarrow \forall v\,[\mathit{Holds}(t, f, v) \leftrightarrow \mathit{Holds}(t+1, f, v)]] \tag{1}$$

Formulas that model a reasoning domain must be written so that fluents are explicitly occluded in situations where they are known to change values, such as in the effects of an action. Instead of adding additional negated occlusion formulas one would like to minimize the number of timepoints where fluents are occluded to implement the default assumption that things do not change without a reason. This minimization is effected through circumscription of the *Occlude* predicate with respect to parts of the theory as detailed in Section 2.5. The result is a solution to the frame problem with fine grained control over which fluents are persistent at which timepoints.

2.2 A TAL Narrative

The TAL language thus includes three predicates, *Holds*, *Occurs*, and *Occlude* in an order-sorted (i.e. supporting a type hierarchy) first-order logic. An encoding of a limited domain of knowledge is called a TAL *narrative*. Consider a very simple reasoning domain involving the flight of a UAV to a new location, expressed by the following narrative:

$$\mathit{Holds}(0, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc1}) \tag{2}$$

$$\mathit{Occurs}(3, 8, \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2})) \tag{3}$$

$$\forall t_1, t_2, u, l\,[t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{fly}(u, l)) \rightarrow \mathit{Holds}(t_2, \mathrm{loc}(u), l) \wedge \forall t\,[t_1 < t \leq t_2 \rightarrow \mathit{Occlude}(t, \mathrm{loc}(u))]] \tag{4}$$

$$\forall t, u\,[\neg \mathit{Occlude}(t+1, \mathrm{loc}(u)) \rightarrow \forall v\,[\mathit{Holds}(t, \mathrm{loc}(u), v) \leftrightarrow \mathit{Holds}(t+1, \mathrm{loc}(u), v)]] \tag{5}$$

Formulas (2) and (3) state that *uav1* is located at *loc1* and then flies to *loc2* between timepoints 3 and 8. The action specification formula (4) occludes the location fluent from persistence during the flight and makes sure it assumes the correct value at the final timepoint of the occurrence interval. Finally, formula (5) instantiates the general persistence formula (1) for the *loc* fluent.

2.3 The High-level Language

The reification of fluents as terms enable the application of regular first-order logic to the problem of reasoning about actions and change over time. But formulas in the resulting theory must necessarily contain predicates and supporting axioms of a technical nature that are not present in any natural language description of the problem domain that one is trying to formalize. TAL provides a high-level narrative description language that helps circumvent this problem through a clear and concise syntax that lifts the abstraction level up above technical details. To differentiate this language from the base logic they are denoted $\mathcal{L}$(ND) (for Language of Narrative Descriptions) and $\mathcal{L}$(FL) (for Language of First-order Logic) respectively. The high-level language $\mathcal{L}$(ND) does not have a proof theory but is seen as a macro language that can be compiled into $\mathcal{L}$(FL) through a translation function *Trans*. Reasoning can then proceed using first-order proof techniques as described in Section 2.6.

Consider the same UAV flight domain introduced above but expressed as a narrative in the high-level language $\mathcal{L}$(ND):

$$[0]\ \mathrm{loc}(\mathrm{uav1}) \hat{=} \mathrm{loc1} \tag{6}$$

$$[3, 8]\ \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2}) \tag{7}$$

$$[t_1, t_2]\ \mathrm{fly}(u, l) \rightsquigarrow R((t_1, t_2]\ \mathrm{loc}(u) \hat{=} l) \tag{8}$$

$$\forall t, u\,[\mathit{Per}(t, \mathrm{loc}(u))] \tag{9}$$

Notice how fluents are associated with values using $\hat{=}$, as in the observation formula (6), and how temporal contexts are indicated simply by preceding formulas with timepoint or interval indications, as in the action occurrence (7). Formula (8) denotes through $\rightsquigarrow$ and the reassignment macro R that the

fly action will cause the location of the UAV to be released from persistence during flight and settle on the destination when the flight is completed. The release from persistence is effected through occlusion of the *loc* fluent, denoted by $X((t_1, t_2]\ \mathrm{loc}(u) \hat{=} l)$, but the domain modeller need not be concerned about occlusion since this mechanism is built into the reassignment macro. Finally, formula (9) will make the location fluent *loc* persistent by default using the macro *Per*.

In addition to the type of formulas illustrated above, a narrative may also include domain and dependency constraints that express conditions that always hold or dependencies that change the values of fluents when triggered by other fluent value changes rather than by an explicit action occurrence. These are powerful features, although we will not make further use of them in this thesis.

2.3.1 The Translation Function

In its full generality, the syntax of $\mathcal{L}$(ND) and its translation into $\mathcal{L}$(FL) is given by the translation function *Trans*. The three $\mathcal{L}$(FL) predicates have a corresponding $\mathcal{L}$(ND) syntax given by:

$$\mathit{Trans}([t]\, f \hat{=} v) \stackrel{\text{def}}{=} \mathit{Holds}(t, f, v)$$
$$\mathit{Trans}([t_1, t_2]\, a) \stackrel{\text{def}}{=} \mathit{Occurs}(t_1, t_2, a)$$
$$\mathit{Trans}(X([t]\, f \hat{=} v)) \stackrel{\text{def}}{=} \mathit{Occlude}(t, f)$$

Action specifications translate into implications with the occurrence of the action during a non-unit time interval as the antecedent and the (possibly conditional and non-deterministic) effects in the conclusion:

$$\mathit{Trans}([t_1, t_2]\, a(\overline{x}) \rightsquigarrow \Phi) \stackrel{\text{def}}{=} \forall t_1, t_2, \overline{x}\, [t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, a(\overline{x})) \rightarrow \mathit{Trans}(\Phi)]$$

The value of a fluent that is occluded at a timepoint is unknown without the presence of additional information. Persistent fluents are associated with persistency formulas, such as (5), that bind the fluent's value to the value at the previous timepoint when not occluded. A durational fluent, on the other hand, resumes a default value when not occluded. An example could be a fluent *noise* that is true while some loud action is being executed but reverts to false otherwise. Fluents are specified as persistent or durational using the *Per* and *Dur* macros:

$$\begin{aligned}\mathit{Trans}(\mathit{Per}(t, f)) \stackrel{\text{def}}{=}\ & \neg \mathit{Occlude}(t+1, f) \rightarrow \\ & \forall v\, [\mathit{Holds}(t, f, v) \leftrightarrow \mathit{Holds}(t+1, f, v)]\end{aligned}$$
$$\mathit{Trans}(\mathit{Dur}(t, f, v)) \stackrel{\text{def}}{=} \neg \mathit{Occlude}(t, f) \rightarrow \mathit{Holds}(t, f, v)$$

The reassignment operator R and interval reassignment operator I ensure both that the fluent assumes the new value and that it is released from any persistence assumptions using the occlusion operator X:

$$\mathit{Trans}(R((t_1, t_2]\,\Phi)) \stackrel{\text{def}}{=} \mathit{Trans}(X((t_1, t_2]\,\Phi)) \wedge \mathit{Trans}([t_2]\,\Phi)$$

$$\mathit{Trans}(R([t]\,\Phi)) \stackrel{\text{def}}{=} \mathit{Trans}(X([t]\,\Phi)) \wedge \mathit{Trans}([t]\,\Phi)$$

$$\mathit{Trans}(I((t_1, t_2]\,\Phi)) \stackrel{\text{def}}{=} \mathit{Trans}(X((t_1, t_2]\,\Phi)) \wedge \mathit{Trans}((t_1, t_2]\,\Phi)$$

$$\mathit{Trans}(I([t]\,\Phi)) \stackrel{\text{def}}{=} \mathit{Trans}(X([t]\,\Phi)) \wedge \mathit{Trans}([t]\,\Phi)$$

Temporal intervals can be open or closed in either end, may be infinite, and can occur inside the occlusion operator X. Their translations are illustrated by some representative examples:

$$\mathit{Trans}([t_1, \infty)\,\Phi) \stackrel{\text{def}}{=} \forall t\,[t_1 \leq t \rightarrow \mathit{Trans}([t]\,\Phi)]$$

$$\mathit{Trans}((t_1, \infty)\,\Phi) \stackrel{\text{def}}{=} \forall t\,[t_1 < t \rightarrow \mathit{Trans}([t]\,\Phi)]$$

$$\mathit{Trans}([t_1, t_2]\,\Phi) \stackrel{\text{def}}{=} \forall t\,[t_1 \leq t \leq t_2 \rightarrow \mathit{Trans}([t]\,\Phi)]$$

$$\mathit{Trans}(X((t_1, t_2]\,\Phi)) \stackrel{\text{def}}{=} \forall t\,[t_1 < t \leq t_2 \rightarrow \mathit{Trans}(X([t]\,\Phi))]$$

The translation function ignores connectives and quantifiers, and temporal contexts are distributed over connectives:

$$\mathit{Trans}(\neg\Phi) \stackrel{\text{def}}{=} \neg\mathit{Trans}(\Phi)$$

$$\mathit{Trans}(\Phi\ C\ \Psi) \stackrel{\text{def}}{=} \mathit{Trans}(\Phi)\ C\ \mathit{Trans}(\Psi) \text{ where } C \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$$

$$\mathit{Trans}(Qv[\Phi]) \stackrel{\text{def}}{=} Qv[\mathit{Trans}(\Phi)] \text{ where } Q \in \{\forall, \exists\}$$

$$\mathit{Trans}([t]\,\neg\Phi) \stackrel{\text{def}}{=} \neg\mathit{Trans}([t]\,\Phi)$$

$$\mathit{Trans}([t]\,\Phi\ C\ \Psi) \stackrel{\text{def}}{=} \mathit{Trans}([t]\,\Phi)\ C\ \mathit{Trans}([t]\,\Psi) \text{ where } C \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$$

$$\mathit{Trans}([t]\,Qv[\Phi]) \stackrel{\text{def}}{=} Qv[\mathit{Trans}([t]\,\Phi)] \text{ where } Q \in \{\forall, \exists\}$$

However, the occlusion operator X needs special attention. For the circumscription of *Occlude* to be computable, as described in Section 2.6, all the fluents that might be influenced by an action must be occluded. This is accomplished by removing negations and replacing disjunctions and existential quantifiers by conjunctions and universal quantifiers:

$$\mathit{Trans}(X([t]\,\neg\Phi)) \stackrel{\text{def}}{=} \mathit{Trans}(X([t]\,\Phi))$$

$$\mathit{Trans}(X([t]\,\Phi\ C\ \Psi)) \stackrel{\text{def}}{=} \mathit{Trans}([t]\,\Phi) \wedge \mathit{Trans}([t]\,\Psi) \text{ where } C \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$$

$$\mathit{Trans}(X([t]\,Qv[\Phi])) \stackrel{\text{def}}{=} \forall v\,[\mathit{Trans}(X([t]\,\Phi))] \text{ where } Q \in \{\forall, \exists\}$$

Finally, some common expressions have shorthand notation. If f is a boolean fluent then f and $\neg f$ are short for $f \mathrel{\hat{=}} \mathit{true}$ and $f \mathrel{\hat{=}} \mathit{false}$, and when several timepoints are ordered in a series $t_1 < t_2 < t_3$, it is short for a conjunction $t_1 < t_2 \wedge t_2 < t_3$.

2.4 Foundational Axioms

Some intuitions about narratives need to be expressed as additional axioms before the domain of interest can be considered formalized. Specifically, these include unique names assumptions, unique value assumptions, and the meaning of functions and relations on timepoints.

When two different objects, such as *loc1* and *loc2*, are used one most probably assumes that they are different, i.e. that the terms satisfy $\mathit{loc1} \neq \mathit{loc2}$. These assumptions have to be made explicit in first-order logic using additional unique names axioms. Since the logic is order-sorted one may still assume that two objects of different types are not the same object, but for any pair of object terms o and o' of the same domain the following axiom is added:

$$o \neq o'$$

The same assumption of uniqueness holds for fluents although two sets of formulas are needed. One with each pair of different fluents f and f', and one where, for each fluent, one or more of its arguments are different:

$$\forall \overline{x}, \overline{y}\,[f(\overline{x}) \neq f'(\overline{y})]$$

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n\,[x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \rightarrow f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n)]$$

Similar unique names axioms are added for action terms:

$$\forall \overline{x}, \overline{y}\,[a(\overline{x}) \neq a'(\overline{y})]$$

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n\,[x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \rightarrow a(x_1, \ldots, x_n) \neq a(y_1, \ldots, y_n)]$$

Unique value axioms, two for each fluent, state that the fluent holds one, and exactly one, value at each timepoint:

$$\forall t, \overline{x}\,[\exists v\,[\mathit{Holds}(t, f(\overline{x}), v)]]$$

$$\forall t, \overline{x}, v_1, v_2\,[v_1 \neq v_2 \rightarrow \neg(\mathit{Holds}(t, f(\overline{x}), v_1) \wedge \mathit{Holds}(t, f(\overline{x}), v_2))]$$

Finally, some of the formulas we have used include the addition function + and the "less than" relation $<$ and "less than or equal" relation $\leq$ on integer time-points. The intention is for them to behave like the mathematical concepts with corresponding notation. It is possible to include an axiomatized theory for this purpose, such as Presburger arithmetic [39], and continue using regular theorem proving with the additional axioms. However, from a practical viewpoint, a more realistic approach is the use of a specialized but incomplete reasoning system, e.g. some variant of a constraint solver as discussed in Section 3.2.1 and carried out in Chapter 4.

2.5 Circumscription Policy

Value changes of fluents that have been specified as persistent are minimized through circumscription, in effect implementing a default assumption of persistence unless an exception was explicitly stated using the *Occlude* predicate. A circumscription policy specifies what predicates to minimize in which formulas, thereby making sure the expected behaviour is obtained. To specify the policy we need to be able to refer to the different sets of formulas that a narrative consists of. For this purpose, let Γ_{fnd} denote the foundational axioms, Γ_{per} persistence formulas, Γ_{obs} fluent value observations, Γ_{domc} domain constraints, Γ_{occ} action occurrences, Γ_{depc} dependency constraints, and Γ_{acs} denote action specifications. The TAL circumscription policy is then given by:

$$\Gamma_{\text{fnd}} \wedge \Gamma_{\text{per}} \wedge \Gamma_{\text{obs}} \wedge \Gamma_{\text{domc}} \wedge \mathit{CIRC}[\Gamma_{\text{occ}}; \mathit{Occurs}] \wedge \mathit{CIRC}[\Gamma_{\text{depc}} \wedge \Gamma_{\text{acs}}; \mathit{Occlude}]$$

The *Occurs* predicate is minimized to remove the possibility of unknown action occurrences. Since action effects entail occlusion they would otherwise prevent proofs of persistence. Note also that persistence formulas (Γ_{per}) are excluded from the minimization of *Occlude* even though they contain the predicate. The contrapositive of the persistence formula (1) illustrates the reason:

$$\forall t\,[\neg\forall v\,[\mathit{Holds}(t, f, v) \leftrightarrow \mathit{Holds}(t+1, f, v)] \rightarrow \mathit{Occlude}(t+1, f)] \tag{10}$$

If (1), which is equivalent to (10), had been included, a fluent value change would have been reason in itself for the occlusion of a fluent, and the minimization of *Occlude* would not have been effective.

2.6 Reasoning in TAL

The goal of the development of Temporal Action Logic is not just to formalize reasoning domains, but also to make it possible for an intelligent agent to use the logic to perform practical reasoning tasks. The availability of reasoning techniques is therefore of great importance, and it is no coincidence that first-order logic was chosen for the base language. Nevertheless, after narratives written in the high-level language $\mathcal{L}(\text{ND})$ have been compiled into the first-order language $\mathcal{L}(\text{FL})$, the circumscription policy is applied and the resulting theory is second-order, where no complete proof methods are possible. By enforcing certain syntactic restrictions on the $\mathcal{L}(\text{ND})$ formulas one can show that the circumscribed predicates *Occurs* and *Occlude* occur only positively in the relevant parts of the theory, Γ_{occ} and $\Gamma_{\text{depc}} \wedge \Gamma_{\text{acs}}$ respectively [8]. Circumscription then reduces to predicate completion [22], which can be computed with relative ease. Any first-order proof method of choice can then be applied.

2.6.1 Natural Deduction

A natural deduction proof of a simple consequence of the UAV narrative introduced in Section 2.2 will serve to illustrate reasoning in TAL. The notation used is that of Copi [31], where each row is made up of a row number (starting from 11 to avoid confusing the row numbers with numbered formulas in previous sections of this chapter), the deduced formula, and a justification of the deduction. We will not explicitly list the natural deduction rules used in the justification but instead assume proof steps that are sufficiently small so as to only require justification in the form of references to the proof rows that the deduction depends on, or a letter indicating a premise P, hypothesis H, tautology T, or a direct consequence of the TAL foundational axioms F. The scope and nesting of hypotheses are indicated by vertical lines in the left margin.

Assume that we would like to prove e.g. that *uav1* will be at *loc2* at timepoint 9. The proof begins by predicate completion of *Occurs* in the action occurrence formula subset Γ_{occ} of the narrative (simply formula (3) above) and of *Occlude* in the action specification and dependency constraint formula subset $\Gamma_{\text{depc}} \wedge \Gamma_{\text{acs}}$ of the narrative (simply formula (4) above), according to the circumscription policy. The resulting premises are:

11 $\mathit{Holds}(0, \text{loc}(\text{uav1}), \text{loc1})$ P

12 $\forall t_1, t_2, a\,[t_1 = 3 \wedge t_2 = 8 \wedge a = \text{fly}(\text{uav1}, \text{loc2}) \leftrightarrow \mathit{Occurs}(t_1, t_2, a)]$ P

13 $\forall t_1, t_2, u, l\,[t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, \text{fly}(u, l)) \rightarrow \mathit{Holds}(t_2, \text{loc}(u), l)]$ P

14 $\forall t, u\,[\exists t_1, t_2, l\,[t_1 < t \leq t_2 \wedge \mathit{Occurs}(t_1, t_2, \text{fly}(u, l))] \leftrightarrow \mathit{Occlude}(t, \text{loc}(u))]$ P

15 $\forall t, u\,[\neg \mathit{Occlude}(t+1, \mathrm{loc}(u)) \rightarrow$
$\forall v\,[\mathit{Holds}(t, \mathrm{loc}(u), v) \leftrightarrow \mathit{Holds}(t+1, \mathrm{loc}(u), v)]]$ P

Note that the action specification formula gave rise to both an action effect formula and the occlusion definition in premises (13) and (14), and that the foundational axioms are not present in this listing.

We proceed to show that the *fly* action takes the UAV to *loc2* between timepoints 3 and 8:

16 $3 < 8 \wedge \mathit{Occurs}(3, 8, \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2})) \rightarrow \mathit{Holds}(8, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2})$ 13

17 $3 < 8$ F

18 $3 = 3 \wedge 8 = 8 \wedge \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2}) = \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2}) \leftrightarrow$ 12
$\mathit{Occurs}(3, 8, \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2}))$

19 $3 = 3 \wedge 8 = 8 \wedge \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2}) = \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2})$ T

20 $\mathit{Occurs}(3, 8, \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2}))$ 18, 19

21 $\mathit{Holds}(8, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2})$ 16, 17, 20

The minimization of *Occlude* has made it possible to prove non-occlusion of the *loc* fluent for timepoints outside the *fly* action occurrence interval. This fact, together with the persistence formula (15), is required to propagate the fluent's value *loc2* from timepoint 8 to timepoint 9:

22 $\neg \mathit{Occlude}(8+1, \mathrm{loc}(\mathrm{uav1})) \rightarrow$
$\mathit{Holds}(8, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2}) \leftrightarrow \mathit{Holds}(8+1, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2})$ 15

23 $8 + 1 = 9$ F

24 $\neg \mathit{Occlude}(9, \mathrm{loc}(\mathrm{uav1})) \rightarrow$
$\mathit{Holds}(8, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2}) \leftrightarrow \mathit{Holds}(9, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2})$ 22, 23

25 $\neg\exists t_1, t_2, l\,[t_1 < 9 \leq t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{fly}(\mathrm{uav1}, l))] \leftrightarrow$
$\neg \mathit{Occlude}(9, \mathrm{loc}(\mathrm{uav1}))$ 14

26 $\exists t_1, t_2, l\,[t_1 < 9 \leq t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{fly}(\mathrm{uav1}, l))]$ H

27 $t_1 < 9 \leq t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{fly}(\mathrm{uav1}, l))$ H

28 $t_1 = 3 \wedge t_2 = 8 \wedge \mathrm{fly}(\mathrm{uav1}, l) = \mathrm{fly}(\mathrm{uav1}, \mathrm{loc2})$ 12, 27

29 $3 < 9 \leq 8$ 27, 28

30 *false* 29, F

31 *false* 26, 27 − 30

32 $\neg\exists t_1, t_2, l\,[t_1 < 9 \leq t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{fly}(\mathrm{uav1}, l))]$ 26 − 31

33 $\neg \mathit{Occlude}(9, \mathrm{loc}(\mathrm{uav1}))$ 25, 32

34 $\mathit{Holds}(8, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2}) \leftrightarrow \mathit{Holds}(9, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2})$ 24, 33

35 $\mathit{Holds}(9, \mathrm{loc}(\mathrm{uav1}), \mathrm{loc2})$ 21, 34

2.6.2 Automated Reasoning

Beside handcrafted proofs, a whole host of automated theorem proving and model theoretic tools are in principle applicable to the first-order narrative. Unfortunately it is far from self-evident that the direct application of theorem provers provides reasoning capabilities that are sufficiently efficient to be of practical use. Especially not when an axiomatization of TAL's integer time structure is used.

In practice we most often rely on a model theoretic tool for evaluating TAL formulas and visualizing their models developed by Kvarnström called VITAL [18]. The tool, which is available online, provides excellent opportunities for experimentation with formalizing reasoning problems as TAL narratives.

This thesis explores a third option, the use of logic programming and constraint solving technology. By introducing further restrictions on $\mathcal{L}(\mathrm{ND})$ formulas it is possible to define a compilation into logic programs that can be efficiently executed by Prolog. Furthermore, the close integration between Prolog and constraint solvers, in the constraint logic programming paradigm, provides a good fit to the TAL logic with its explicit time structures. This topic is explored in Chapter 4 and Chapter 5.

Chapter 3

Deductive Planning

Deductive planning requires modifications to the Temporal Action Logic presented in Chapter 2. When action occurrences are specified using the first-order *predicate Occurs* there is no mechanism that allows the addition of new occurrences through a deductive proof process. Hypothesizing new instances of this predicate given a goal would require abductive techniques or the use of some special-purpose planning algorithm such as that employed by TALplanner [7]. This chapter introduces a *deductive* mechanism through the reification of action occurrences as terms in the language that can be reasoned with and quantified over. In addition, TAL is extended with the concept of interval occlusion that makes the use of temporal constraint formalisms easier. Constraint solvers eliminate the need for an axiomatization of the integer timeline for the types of temporal reasoning needed in our applications to deductive planning. A hand crafted proof illustrates how reified actions and interval occlusion make planning possible, although still not practical.

3.1 Reified Action Occurrences

A deductive planning goal Φ is usually expressed as the existence of a plan that satisfies Φ. In TAL we would add an explicit timepoint at which the goal is satisfied:

$$\exists t, p\,[\Phi(t, p)]$$

While the conclusion in proving the above is the *existance* of a plan, the actual plan can be retrieved from that proof. However, since plans are composed of action occurrences that are denoted using the TAL *predicate Occurs*, there is

no way to specify a (first-order) existential quantifier over plans in TAL. This problem can be resolved by reifying action occurrences as terms, which can be quantified over. Doing so necessitates the introduction of a *Member* predicate, to group action occurrence terms into sets, or plans, and the addition of a new action occurrence set argument to the other two TAL predicates *Holds* and *Occlude*, so that we can talk about e.g. a fluent's value in the context of a specific plan.

We start by introducing a function *occ* that replaces action occurrences $Occurs(t_1, t_2, a)$ by terms of the form $occ(t_1, t_2, a)$. A set of action occurrences is a set of such terms. The temporal ordering of the actions themselves is given by their relations to the explicit time line, determined by the t_1 and t_2 timepoint arguments. In consequence, the order of the action occurrence terms in the set is not important, which means that its behaviour is just like a regular mathematical set.

Additionally, a vital property is incompleteness of the action occurrence set. If a reasoning problem involves a fully specified set of action occurrences, then the problem corresponds to prediction or postdiction of the effects of executing the actions or the conditions that hold when executing those actions. If a reasoning problem involves an under-specified (or empty) set of action occurrences, then the problem often involves the addition of new action occurrences to the occurrence set to satisfy goal constraints that were expressed as part of the problem. This constitutes a process of deductive planning.

We introduce a composition function term $cons(a, p)$ to add an individual action occurrence term a to an action occurrence set (or *p*lan) p, and the term *nil* to represent the empty set. A predicate *Member* is associated with a simple axiomatization of set membership:

$$\forall x\,[\neg Member(x, nil)]$$
$$\forall x, y, s\,[Member(x, cons(y, s)) \leftrightarrow x = y \vee Member(x, s)]$$

A new *Occurs* predicate with an additional action occurrence set argument is defined as:

$$\forall t_1, t_2, a, p\,[Occurs(t_1, t_2, a, p) \leftrightarrow Member(occ(t_1, t_2, a), p)]$$

Lastly, an action occurrence argument is added to the other two $\mathcal{L}$(FL) predicates *Holds* and *Occlude*. The meaning of a narrative is preserved by introducing a new universally quantified action occurrence set variable p in each of the narrative formulas. E.g., the *fly* action specification formula from the example narrative in Chapter 2, repeated here:

$$\forall t_1, t_2, u, l\,[t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{fly}(u, l)) \rightarrow \\ \mathit{Holds}(t_2, \mathrm{loc}(u), l) \wedge \forall t\,[t_1 < t \leq t_2 \rightarrow \mathit{Occlude}(t, \mathrm{loc}(u))]]$$

would receive an additional universally quantified action occurrence set argument p simply by adding it to each of the $\mathcal{L}$(FL) predicates:

$$\forall t_1, t_2, u, l, p\,[t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{fly}(u, l), p) \rightarrow \\ \mathit{Holds}(t_2, \mathrm{loc}(u), l, p) \wedge \forall t\,[t_1 < t \leq t_2 \rightarrow \mathit{Occlude}(t, \mathrm{loc}(u), p)]]$$

Using the same variable p throughout the formula ensures that dependencies on action occurrences are preserved just as if the occurrences had been specified separately using the old *Occurs* predicate. Then, by universal quantification over p, the formula is made applicable for any set of action occurrences that one might want to reason about.

3.2 Interval Occlusion

The TAL *Occlude* predicate determines at which timepoints a fluent may change value. Negated occlusion is, accordingly, the property of a fluent *not* being allowed to change, i.e. the property of persistence. Proofs of persistence between two timepoints are very frequent in the context of planning and the most straightforward way of proving persistence involves deducing $\neg \mathit{Occlude}(t_i, f)$ for the fluent of interest f and for all intermediate timepoints t_i. A fluent persistency formula is then applied to propagate the fluent's value from one timepoint to the other.

However, when the number of intermediate timepoints is not known the simple proof sketched above is not applicable. This is the case e.g. when only qualitative timepoints are used, or when a least commitment strategy concerning the timepoints and orderings of action occurrences is strived for. In such cases it is more useful to look at all the intermediate timepoints as an interval and persistence over all the timepoints as *interval persistence*. As a first step in this direction we introduce the *interval occlusion* predicate $\mathit{Occlude}(t_1, t_2, f)$ (or $\mathit{Occlude}(t_1, t_2, f, p)$ if action occurrences are reified). The intended meaning is that fluent f is interval occluded over $(t_1, t_2]$ iff it is occluded at some timepoint in that interval. Conversely, if we manage to prove that fluent f is *not* interval occluded for t_1 and t_2, then we know that its value will persist. Formally, we define interval occlusion in terms of the regular timepoint occlusion as:

$$\forall t_1, t_2, f\,[\mathit{Occlude}(t_1, t_2, f) \leftrightarrow \exists t\,[t_1 < t \leq t_2 \wedge \mathit{Occlude}(t, f)]] \qquad (1)$$

In Appendix B.1 we prove that (1) together with the persistence formula for timepoint occlusion entails an interval persistence formula:

$$\forall t_1, t_2, f\,[\neg \mathit{Occlude}(t_1, t_2, f) \rightarrow \forall t\,[t_1 < t \leq t_2 \rightarrow \forall v\,[\mathit{Holds}(t-1, f, v) \leftrightarrow \mathit{Holds}(t, f, v)]]] \quad (2)$$

Note that even if a fluent is interval occluded over a given interval, it is not necessarily occluded in all sub-intervals. However, if the fluent is interval persistent over the interval it must also be persistent in all sub-intervals. As shown in Appendix B.2, formulas (1) and (2) entail the following formula:

$$\forall t_1, t_2, f\,[\neg \mathit{Occlude}(t_1, t_2, f) \rightarrow \forall v\,[\mathit{Holds}(t_1, f, v) \leftrightarrow \mathit{Holds}(t_2, f, v)]] \quad (3)$$

Using (3), the truth value of a fluent can be made to "jump" any number of timepoints in a single proof step. This technique is essential in the translation to constraint logic programs described in Chapter 4.

3.2.1 Temporal Constraint Formalisms

Another essential step towards efficient proof procedures for TAL in the context of only qualitative and partial knowledge of temporal relations is the introduction of temporal constraint propagation techniques. While the $\mathit{Holds}(t, f, v)$ predicate is defined on timepoints, both the $\mathit{Occurs}(t_1, t_2, a)$ and the new $\mathit{Occlude}(t_1, t_2, f)$ predicates are defined over intervals characterized by their end-points t_1 and t_2. Furthermore, considering persistence over entire intervals was found to result in conveniently simple persistence proofs. These facts suggest that the well-known interval algebra formalism introduced by Allen [1], which adopts the interval as a primitive temporal object, might be useful. Allen shows that thirteen primitive relations exhaust all possible qualitative relations between two intervals, and the algebra is complete for reasoning with them, even in the context of incomplete information. Allen also notes that intervals can be represented by their end-points and primitive relations by ordering conditions on these end-points.

Thornton et. al. [38] go on to show how their *end-point ordering* model can be used to express the qualitative interval algebra in a quantitative constraint solver with finite domains. They recognize that all solutions to an interval algebra problem must be possible to express by different integer instantiations of the end-points that satisfy the ordering conditions. Furthermore, if there are m intervals in the problem, $2m$ is an upper bound on the number of different integers needed. Hence, an interval algebra network can be encoded as a finite domain constraint satisfaction problem where variables represent interval end-points and belong to the range $1 \ldots 2m$.

We have chosen to work with a finite domain constraint solver and the end-point ordering encoding of the interval algebra, and Chapter 4 provides details on how this is used to implement temporal relations in TAL narratives

expressed as logic programs. Although, other temporal constraint formalisms than the interval algebra are certainly possible. We have in fact experimented with general temporal constraint networks [28], a unifying framework that provides both intervals and timepoints as primitives and both qualitative and quantitative relations between them. Such networks are also associated with complete reasoning procedures of the same (exponential) computational complexity. However, the simpler interval algebra end-point ordering model uses only the $<$, $=$, and $>$ relations between timepoints, requiring no extensions of TAL. General temporal constraint networks support more complex relations whose description in TAL require some form of semantic attachment. Moreover, the upper bound on the domains of timepoint variables, required when working with the finite domain constraint solver, enforces an upper bound on the number of actions in action occurrence sets. This follows from the fact that each action occurs over an interval that is added to the interval algebra graph. Such a bound prevents infinite sequences of actions that can in certain domains keep the planner busy, preventing it from finding a solution. Completeness can be achieved through an iterative deepening of this bound. In practice, the finite domain constraint solver implementation is also considerably faster.

3.3 A Constructive Plan Existence Proof

The above extensions realize planning as deduction. Consider, e.g., the following narrative in $\mathcal{L}$(FL) that extends the narrative in Section 2.2 towards a logistics scenario for UAVs. A location-valued fluent *loc(object)* and a boolean fluent *carrying(uav,crate)* are used to represent the location of an object (either a UAV or a crate) and the fact that a UAV is carrying (or not) a crate respectively. Two actions, *fly(uav,location)* and *attach(uav,crate)*, can be used for flying a UAV to a location and to instruct a UAV to attach its winch to a crate. For the purpose of this example, the narrative formulas make use of interval occlusion. A principled way of introducing interval occlusion in a narrative expressed using regular occlusion is detailed in Section 4.3.

Listing the narrative formulas as premises in the natural deduction style notation introduced in Section 2.6.1 results in the following rows:

4 $\forall x\,[\neg \mathit{Member}(x, \mathit{nil})]$ P

5 $\forall x, y, s\,[\mathit{Member}(x, \mathit{cons}(y, s)) \leftrightarrow x = y \vee \mathit{Member}(x, s)]$ P

6 $\forall t_1, t_2, a, p\,[\mathit{Occurs}(t_1, t_2, a, p) \leftrightarrow \mathit{Member}(\mathit{occ}(t_1, t_2, a), p)]$ P

7 $\forall t_1, t_2, o, p\,[t_1 < t_2 \wedge \neg \mathit{Occlude}(t_1, t_2, \mathrm{loc}(o), p) \rightarrow$
$\forall v\,[\mathit{Holds}(t_1, \mathrm{loc}(o), v, p) \leftrightarrow \mathit{Holds}(t_2, \mathrm{loc}(o), v, p)]]$ P

8 $\forall t_1, t_2, u, c, p\,[t_1 < t_2 \wedge \neg \mathit{Occlude}(t_1, t_2, \text{carrying}(u, c), p) \rightarrow$
$\forall v\,[\mathit{Holds}(t_1, \text{carrying}(u, c), v, p) \leftrightarrow \mathit{Holds}(t_2, \text{carrying}(u, c), v, p)]]$ P

9 $\forall t_1, t_2, u, l, p\,[t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, \text{fly}(u, l), p) \rightarrow \mathit{Holds}(t_2, \text{loc}(u), l, p)]$ P

10 $\forall t_1, t_2, u, c, l, p\,[t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, \text{attach}(u, c), p) \wedge$
$\mathit{Holds}(t_1, \text{loc}(c), l, p) \wedge \mathit{Holds}(t_1, \text{loc}(u), l, p) \rightarrow$
$\mathit{Holds}(t_2, \text{carrying}(u, c), \mathit{true}, p) \wedge \neg \mathit{Holds}(t_2, \text{loc}(c), l, p)]$ P

11 $\forall t_1, t_2, t_3, t_4, u, c, l, p\,[t_2 > t_3 \wedge t_1 < t_4 \wedge \mathit{Occurs}(t_3, t_4, \text{attach}(u, c), p) \wedge$
$\mathit{Holds}(t_3, \text{loc}(c), l, p) \wedge \mathit{Holds}(t_3, \text{loc}(u), l, p) \leftrightarrow$
$\mathit{Occlude}(t_1, t_2, \text{carrying}(u, c), p)]$ P

12 $\forall t_1, t_2, t_3, t_4, u, c, l, p\,[t_2 > t_3 \wedge t_1 < t_4 \wedge \mathit{Occurs}(t_3, t_4, \text{attach}(u, c), p) \wedge$
$\mathit{Holds}(t_3, \text{loc}(c), l, p) \wedge \mathit{Holds}(t_3, \text{loc}(u), l, p) \leftrightarrow$
$\mathit{Occlude}(t_1, t_2, \text{loc}(c), p)]$ P

13 $\forall p\,[\mathit{Holds}(0, \text{loc}(\text{uav1}), \text{base}, p)]$ P

14 $\forall p\,[\mathit{Holds}(0, \text{loc}(\text{crate1}), \text{loc1}, p)]$ P

15 $\forall u, c, p\,[\mathit{Holds}(0, \text{carrying}(u, c), \text{false}, p)]$ P

Given this narrative one would like to find constructive proofs of plan existence queries. As an example, it should be possible to prove that there exists some set of action occurrences p (a solution plan) that results in *uav1* carrying *crate1* at some future timepoint t. The main goal of the proof can then be expressed as the existence of such a timepoint and plan for which the goal holds:

$\exists t, p\,[\mathit{Holds}(t, \text{carrying}(\text{uav1}, \text{crate1}), \mathit{true}, p)]$

The exact value of timepoint t can be unspecified as long as the actions in p will fit between timepoints 0 and t. A single intermediate timepoint t_k suffices for the purposes of this proof:

16 $0 < t_k < t$ P

One way of proving the main goal is by instantiating the *attach* action specification with *uav1*, *crate1*, and t:

17 $\forall t_1, l, p\,[t_1 < t \wedge \mathit{Occurs}(t_1, t, \text{attach}(\text{uav1}, \text{crate1}), p) \wedge$
$\mathit{Holds}(t_1, \text{loc}(\text{crate1}), l, p) \wedge \mathit{Holds}(t_1, \text{loc}(\text{uav1}), l, p) \rightarrow$
$\mathit{Holds}(t, \text{carrying}(\text{uav1}, \text{crate1}), \mathit{true}, p)]$ 10

If the implication antecedent of Row 17 can be proven, the goal will follow from Modus Ponens. Call the four antecedent conjuncts Subgoal 1 – 4. Subgoal 1 can be satisfied immediately:

18 $t_k < t$ 16

Subgoal 2 requires the use of the definition of *Occurs*. This creates an action occurrence set that includes an occurrence of the *attach* action and a new unknown plan tail p_1:

19 $occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})) = occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})) \vee$
$Member(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1) \rightarrow$
$Member(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$ 5

20 $occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})) = occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1}))$ T

21 $Member(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$ 19, 20

22 $Occurs(t_k, t, \text{attach}(\text{uav1}, \text{crate1}),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$ 6, 21

23 $\forall p_1\, [Occurs(t_k, t, \text{attach}(\text{uav1}, \text{crate1}),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))]$ 22

One way of satisfying Subgoal 3, $Holds(t_1, \text{loc}(\text{crate1}), l, p)$, would be to use Row 14 that specifies the location of *crate1* in the initial state. But t_1 would then be instantiated to 0, which would leave the UAV no time to fly there and satisfy Subgoal 4. Instead, we use the persistence of the *loc* fluent to show that *crate1* remains at *loc1* from 0 to an intermediate timepoint t_k. An automated theorem prover, without foresight, would try both the initial state fact and the persistence formula, either simultaneously, or through backtracking. The persistence formula in Row 7 is reformulated as an implication:

24 $\forall v, p_1\, [0 < t_k \wedge$
$\neg Occlude(0, t_k, \text{loc}(\text{crate1}),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)) \wedge$
$Holds(0, \text{loc}(\text{crate1}), v, cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)) \rightarrow$
$Holds(t_k, \text{loc}(\text{crate1}), v, cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))]$ 7

If the three antecedent conjuncts of Row 24 are referred to as Subgoal 3.1 – 3.3, then Subgoal 3.1 is given by:

25 $0 < t_k$ 16

In order to succeed with the proof of non-occlusion in Subgoal 3.2 it is necessary to close the action occurrence argument p_1 with respect to any unknown *attach* actions. The TAL circumscription policy usually closes action occurrences through circumscription of the *Occurs* predicate. The same effect can be achieved with reified action occurrences by recording assumptions made about the plan argument p. Such assumptions can be proven after instantiating unbound plan variables to *nil* when no new occurrences need to be added

to the final plan. The proof of Subgoal 3.2 records the assumption that p_1 contains no occurrences of *attach* in Row 26. Non-occlusion of the *loc* fluent can be proved using its definition from Row 12. Assuming that the conditions for occlusion are satisfied, in Row 28, leads to a contradiction. It would either have meant that the persistence interval $(0, t_k]$ overlaps the *attach* action interval $(t_k, t]$, disproved in Row 29 – Row 34, or that there are some other *attach* actions, which we assumed false as shown by Row 35 – Row 37:

26	$\neg\exists t_1, t_2, u, c\,[\mathit{Member}(\mathit{occ}(t_1, t_2, \text{attach}(u, c)), p_1)]$	H
27	$\neg(t_k > t_3 \wedge 0 < t_4 \wedge$ $\mathit{Occurs}(t_3, t_4, \text{attach}(u, c), \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)) \wedge$ $\mathit{Holds}(t_3, \text{loc}(\text{crate1}), l, \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)) \wedge$ $\mathit{Holds}(t_3, \text{loc}(u), l, \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))) \rightarrow$ $\neg\mathit{Occlude}(0, t_k, \text{loc}(\text{crate1}),$ $\mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$	12
28	$t_k > t_3 \wedge 0 < t_4 \wedge$ $\mathit{Occurs}(t_3, t_4, \text{attach}(u, c), \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)) \wedge$ $\mathit{Holds}(t_3, \text{loc}(\text{crate1}), l, \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)) \wedge$ $\mathit{Holds}(t_3, \text{loc}(u), l, \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$	H
29	$\mathit{Member}(\mathit{occ}(t_3, t_4, \text{attach}(u, c)),$ $\mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$	6, 28
30	$\mathit{occ}(t_3, t_4, \text{attach}(u, c)) = \mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})) \vee$ $\mathit{Member}(\mathit{occ}(t_3, t_4, \text{attach}(u, c)), p_1)$	5, 29
31	$\mathit{occ}(t_3, t_4, \text{attach}(u, c)) = \mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1}))$	H
32	$t_3 = t_k \wedge t_4 = t \wedge u = \text{uav1} \wedge c = \text{crate1}$	31
33	$t_k > t_k \wedge 0 < t$	28, 32
34	$\bot$	33, F
35	$\mathit{Member}(\mathit{occ}(t_3, t_4, \text{attach}(u, c)), p_1)$	H
36	$\exists t_1, t_2, u, c\,[\mathit{Member}(\mathit{occ}(t_1, t_2, \text{attach}(u, c)), p_1)]$	35
37	$\bot$	26, 36
38	$\bot$	$30, 31 - 34, 35 - 37$
39	$\neg(t_k > t_3 \wedge 0 < t_4 \wedge$ $\mathit{Occurs}(t_3, t_4, \text{attach}(u, c), \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)) \wedge$ $\mathit{Holds}(t_3, \text{loc}(\text{crate1}), l, \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)) \wedge$ $\mathit{Holds}(t_3, \text{loc}(u), l, \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1)))$	$28 - 38$

40 | $\neg Occlude(0, t_k, \text{loc}(\text{crate1}),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$ 27, 39

41 $\neg\exists t_1, t_2, u, c\,[Member(occ(t_1, t_2, \text{attach}(u, c)), p_1)] \rightarrow$
$\neg Occlude(0, t_k, \text{loc}(\text{crate1}),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$ $26 - 40$

42 $\forall p_1\,[\neg\exists t_1, t_2, u, c\,[Member(occ(t_1, t_2, \text{attach}(u, c)), p_1)] \rightarrow$
$\neg Occlude(0, t_k, \text{loc}(\text{crate1}),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))]$ 41

Knowledge about the initial state in Row 14 is then used to satisfy Subgoal 3.3 with v instantiated to *loc1*:

43 $\forall p_1\,[Holds(0, \text{loc}(\text{crate1}), \text{loc1}, cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))]$ 14

Subgoal 3 can now be completed, although it is still qualified by the assumption on occurrences of *attach*:

44 | $\neg\exists t_1, t_2, u, c\,[Member(occ(t_1, t_2, \text{attach}(u, c)), p_1)]$ H

45 | $Holds(t_k, \text{loc}(\text{crate1}), true,$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))$ $24, 25, 42, 44, 43$

46 $\forall p_1\,[\neg\exists t_1, t_2, u, c\,[Member(occ(t_1, t_2, \text{attach}(u, c)), p_1)] \rightarrow$
$Holds(t_k, \text{loc}(\text{crate1}), \text{loc1},$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})), p_1))]$ $44 - 45$

The *fly* action specification in Row 9 can be used to prove Subgoal 4. The definition of *Occurs* is used to add a *fly* action to the plan, resulting in a new plan tail p_2. The new action causes the UAV to end up at the location of the crate, thereby satisfying the last condition for attaching it:

47 $0 < t_k \wedge Occurs(0, t_k, \text{fly}(\text{uav1}, \text{loc1}),$
$cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2))) \rightarrow$
$Holds(t_k, \text{loc}(\text{uav1}), \text{loc1}, cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2)))$ 9

48 $occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})) = occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})) \vee$
$Member(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2) \rightarrow$
$Member(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2))$ 5

49 $occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})) = occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1}))$ T

50 $Member(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})),$
$cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2))$ $48, 49$

51 $occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})) = occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})) \vee$
$Member(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})),$
$\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2)) \rightarrow$
$Member(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})),$
$\quad cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$\quad\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2)))$ 5

52 $Member(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})),$
$\quad cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$\quad\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2)))$ 50, 51

53 $Occurs(0, t_k, \text{fly}(\text{uav1}, \text{loc1}),$
$\quad cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$\quad\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2)))$ 6, 52

54 $Holds(t_k, \text{loc}(\text{uav1}), \text{loc1},$
$\quad cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$\quad\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2)))$ 25, 47, 53

55 $\forall p_2\,[Holds(t_k, \text{loc}(\text{uav1}), \text{loc1},$
$\quad cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$\quad\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2)))]$ 54

At this point, Subgoal 1 – 4 in the antecedent of Row 17 have been proved for the following instantiations of the universally quantified variables:

$t_1 = t_k$
$l = \text{loc1}$
$p_1 = cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2)$
$p = cons(occ(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), p_2))$

The qualification on Subgoal 3 can be satisfied by choosing a plan tail p_2 that does not contain any *attach* actions. The most obvious choice is $p_2 = nil$:

56 $\forall x, y, s\,[\neg(x = y) \wedge \neg Member(x, s) \rightarrow \neg Member(x, cons(y, s))]$ 5

57 $\neg(occ(t_1, t_2, \text{attach}(u, c)) = occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})))$ F

58 $\neg Member(occ(t_1, t_2, \text{attach}(u, c)), nil)$ 4

59 $\forall t_1, t_2, u, c\,[\neg Member(occ(t_1, t_2, \text{attach}(u, c)),$
$\quad\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), nil))]$ 56, 57, 58

60 $\neg\exists t_1, t_2, u, c\,[Member(occ(t_1, t_2, \text{attach}(u, c)),$
$\quad\quad cons(occ(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), nil))]$ 59

This completes the proof of the main goal:

61 $\mathit{Holds}(t, \text{carrying}(\text{uav1}, \text{crate1}), \mathit{true},$
$\quad \mathit{cons}(\mathit{occ}(t_k, t, \text{attach}(\text{uav1}, \text{crate1})),$
$\quad\quad \mathit{cons}(\mathit{occ}(0, t_k, \text{fly}(\text{uav1}, \text{loc1})), \mathit{nil})))$ 17, 18, 23, 46, 55, 60

62 $\exists t, p\, [\mathit{Holds}(t, \text{carrying}(\text{uav1}, \text{crate1}), \mathit{true}, p)]$ 61

By extracting the plan from the action occurrence set one can utilize deductive proofs for planning. The above proof was one of many possible constructive proofs of the goal. Through the application of an automated theorem prover that considers all the possibilities in the different choice points in the proof process one obtains a complete planner. However, even the trivial planning problem solved above had a relatively long and complicated proof. Direct application of automated theorem provers to an $\mathcal{L}$(FL) narrative would not result in a practical planning system. Clearly, more work needs to be done before deductive planning becomes truly useful in practice. This is the topic of the next chapter.

Chapter 4

Compiling TAL into Logic Programs

We introduce a four step translation from $\mathcal{L}$(ND) narratives into constraint logic programs that makes use of the reified action occurrences, interval occlusion, and temporal constraint formalisms described in Chapter 3. The four steps are as follows:

(Section 4.2)	Compile the TAL narrative from the surface language $\mathcal{L}$(ND) to the first-order logic language $\mathcal{L}$(FL).
(Section 4.3)	Replace timepoint occlusion in formulas by interval occlusion.
(Section 4.4)	Rewrite all formulas into Horn clause form.
(Section 4.6)	Encode the Horn clauses as Prolog clauses, constraint handling rules, and finite domain constraints.

The resulting programs are efficiently evaluated using Prolog and can be used for both deductive planning and reasoning. The compilation provides a means for the application of deductive planning in TAL to solving practical reasoning problems such as those encountered by our UAV research platform.

Any translation to constraint logic programs is bound to be partial since the source language has the expressivity of first-order logic while the target language is limited to Horn clause form and the constraint formalisms used. Restrictions are therefore set up to isolate the class of narratives of interest. To prevent the implications of this from being lost in the technical details of the translation process we conclude this chapter with a summary and discussion in Section 4.7.

4.1 $\mathcal{L}$(ND) Narrative

The first restriction is to narratives without domain or dependency constraints. Such narratives will only consist of action specifications that detail the conditions and effects of actions, persistence statements that specify which fluents retain their value over time, and observation statements that describe the initial state of the world. Action specifications are assumed to be of the form:

$$[t_1, t_2]\ a(\overline{w}) \rightsquigarrow \forall \overline{x}\,[P(\overline{y}) \rightarrow E(\overline{z})] \tag{1}$$

where $\overline{y}, \overline{z} \subseteq \overline{w} \cup \overline{x}$. $P(\overline{y})$ is a conjunction of conditions on fluent values, each having the form:

$$[t_1, t_2]\ f(\overline{y}) \hat{=} v$$

where the end timepoint t_2 is optional and the value v could be one of the variables in $\overline{w} \cup \overline{x}$, or a constant, including the boolean constants *true* and *false*. $E(\overline{z})$ is a conjunction of reassignments of fluent values, using the reassignment macro R, each of the form:

$$R((t_1, t_2]\ f(\overline{z}) \hat{=} v)$$

where $\hat{=}$ can be replaced by $\hat{\neq}$ and, again, v belongs to $\overline{w} \cup \overline{x}$ or is a constant. Fluents will usually be persistent, as specified by persistence statements of the form:

$$\forall t, \overline{x}\,[Per(t, f(\overline{x}))]$$

Finally, the values of fluents at the start of the planning process are (completely) specified using observations at timepoint zero of the form:

$$[0]\ f \hat{=} v$$

where v is a constant. In order to make these general forms more concrete they will be accompanied by a running example consisting of the UAV logistics scenario fragment from Section 3.3. Again, the example consists of an action *attach*, used by a UAV to attach a crate that can then be moved to another location. The action specification involves a location-valued fluent *loc(object)*, representing the location of an object, and a boolean-valued fluent *carrying(uav,crate)* that is true whenever the UAV given as the first argument is carrying the crate in the second argument. The *attach* action will make the *carrying* fluent true and the value of the *loc* fluent unspecified given that the UAV remains in the same location as the crate during the execution of the action. The $\mathcal{L}$(ND) specification is given by:

$$[t_1, t_2]\ \text{attach}(u, c) \rightsquigarrow$$
$$\forall l\ [[t_1]\ \text{loc}(c) \hat{=} l \wedge [t_1, t_2]\ \text{loc}(u) \hat{=} l \rightarrow$$
$$R((t_1, t_2]\ \text{carrying}(u, c)) \wedge R((t_1, t_2]\ \neg\text{loc}(c) \hat{=} l)]$$

Both *carrying* and *loc* are persistent fluents:

$$\forall t, o\ [Per(t, \text{loc}(o))]$$
$$\forall t, u, c\ [Per(t, \text{carrying}(u, c))]$$

Finally, an initial state is set up. We introduce a fluent *pos* that can be used to refer to locations using coordinates, e.g. as follows:

$$[0]\ \text{loc}(\text{uav1}) \hat{=} \text{pos}(300, 90)$$
$$[0]\ \text{loc}(\text{crate1}) \hat{=} \text{pos}(300, 90)$$
$$[0]\ \neg\text{carrying}(\text{uav1}, \text{crate1})$$

4.2 Translation from $\mathcal{L}$(ND) to $\mathcal{L}$(FL)

The first step in translating the above $\mathcal{L}$(ND) narrative to an executable logic program is to use the transformation function *Trans* to compile the narrative into the first-order logic language $\mathcal{L}$(FL). Using the definition of *Trans* given in Section 2.3.1, augmented with the reified action occurrences argument p, the action specifications (1) are translated into:

$$\forall t_1, t_2, \overline{w}, p\ [t_1 < t_2 \wedge Occurs(t_1, t_2, a(\overline{w}), p) \rightarrow \forall \overline{x}\ [P(\overline{y}) \rightarrow E(\overline{z})]] \tag{2}$$

Each condition in the conjunction $P(\overline{y})$ has been translated into:

$$Holds(t_1, f(\overline{y}), v, p)$$

or to $\forall t\,[t_1 \leq t \leq t_2 \rightarrow Holds(t, f(\overline{y}), v, p)]$ if the condition was specified to hold throughout the entire action interval by supplying an end timepoint t_2. Each reassignment in the conjunction $E(\overline{z})$ is translated into:

$$(\neg)Holds(t_2, f(\overline{z}), v, p) \wedge \forall t\ [t_1 < t \leq t_2 \rightarrow Occlude(t, f(\overline{z}), p)]$$

Persistence statements are instantiations of the general persistence formula for a given fluent f, with the addition of the action occurrence argument:

$$\forall t, \overline{x}, p\ [\neg Occlude(t + 1, f(\overline{x}), p) \rightarrow$$
$$\forall v\ [Holds(t, f(\overline{x}), v, p) \leftrightarrow Holds(t + 1, f(\overline{x}), v, p)]]$$

Each initial fluent value observation becomes:

$$\forall p\ [Holds(0, f, v, p)]$$

Finally, the *Occurs* predicate for reified action occurrences was defined in Section 3.1 as:

$$\forall t_1, t_2, a, p\,[Occurs(t_1, t_2, a, p) \leftrightarrow Member(occ(t_1, t_2, a), p)] \tag{3}$$

Returning to the example from the UAV logistics narrative fragment, its action specification, persistence statements, and initial state observations are translated into the following set of formulas:

$$\begin{aligned}
&\forall t_1, t_2, u, c, l, p\,[t_1 < t_2 \wedge Occurs(t_1, t_2, \text{attach}(u, c), p) \rightarrow \\
&\quad Holds(t_1, \text{loc}(c), l, p) \wedge \forall t\,[t_1 \leq t \leq t_2 \rightarrow Holds(t, \text{loc}(u), l, p)] \rightarrow \\
&\qquad Holds(t_2, \text{carrying}(u, c), \text{true}, p) \wedge \\
&\qquad \forall t\,[t_1 < t \leq t_2 \rightarrow Occlude(t, \text{carrying}(u, c), p)] \wedge \\
&\qquad \neg Holds(t_2, \text{loc}(c), l, p) \wedge \\
&\qquad \forall t\,[t_1 < t \leq t_2 \rightarrow Occlude(t, \text{loc}(c), p)]] \\
&\forall t, x, p\,[\neg Occlude(t + 1, \text{loc}(x), p) \rightarrow \\
&\quad \forall v\,[Holds(t, \text{loc}(x), v, p) \leftrightarrow Holds(t + 1, \text{loc}(x), v, p)]] \\
&\forall t, u, c, p\,[\neg Occlude(t + 1, \text{carrying}(u, c), p) \rightarrow \\
&\quad \forall v\,[Holds(t, \text{carrying}(u, c), v, p) \leftrightarrow Holds(t + 1, \text{carrying}(u, c), v, p)]] \\
&\forall p\,[Holds(0, \text{loc}(\text{uav1}), \text{pos}(300, 90), p)] \\
&\forall p\,[Holds(0, \text{loc}(\text{crate1}), \text{pos}(300, 90), p)] \\
&\forall p\,[Holds(0, \text{carrying}(\text{uav1}, \text{crate1}), \text{false}, p)]
\end{aligned}$$

4.3 Introducing Interval Occlusion

The second step consists of obtaining a definition of timepoint occlusion specific to the given narrative and then replacing it by an equivalent definition of interval occlusion. To accomplish this we split the action specification formulas, each of the form (2), into fluent value reassignments and occlusion formulas. The occlusion predicates are then collected into a single *Occlude* predicate using the equivalence:

$$\Phi(f_1) \wedge \cdots \wedge \Phi(f_n) \equiv \forall x\,[(x = f_1 \vee \cdots \vee x = f_n) \rightarrow \Phi(x)] \tag{4}$$

The resulting form is:

$$\forall t_1, t_2, \overline{w}, \overline{x}, p\,[t_1 < t_2 \wedge Occurs(t_1, t_2, a(\overline{w}), p) \wedge P(\overline{y}) \rightarrow E'(\overline{z})] \tag{5}$$

$$\begin{aligned}
\forall t, f, t_1, t_2, \overline{w}, \overline{x}, p\,[&t_1 < t \leq t_2 \wedge Occurs(t_1, t_2, a(\overline{w}), p) \wedge P(\overline{y}) \wedge \\
&(f = f_1(\overline{z}) \vee \cdots \vee f = f_n(\overline{z})) \rightarrow \\
&Occlude(t, f, p)]
\end{aligned} \tag{6}$$

where E' are the reassignments without the occlusion sub-formulas. Applying this rewrite to the example *attach* action specification produces:

$$\begin{aligned}
&\forall t_1, t_2, u, c, l, p\,[t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{attach}(u, c), p) \wedge \\
&\qquad \mathit{Holds}(t_1, \mathrm{loc}(c), l, p) \wedge \forall t\,[t_1 \leq t \leq t_2 \rightarrow \mathit{Holds}(t, \mathrm{loc}(u), l, p)] \rightarrow \\
&\qquad\qquad \mathit{Holds}(t_2, \mathrm{carrying}(u, c), \mathrm{true}, p) \wedge \neg \mathit{Holds}(t_2, \mathrm{loc}(c), l, p)] \\
&\forall t, f, t_1, t_2, u, c, l, p\,[t_1 < t \leq t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{attach}(u, c), p) \wedge \\
&\qquad \mathit{Holds}(t_1, \mathrm{loc}(c), l, p) \wedge \forall t\,[t_1 \leq t \leq t_2 \rightarrow \mathit{Holds}(t, \mathrm{loc}(u), l, p)] \wedge \\
&\qquad (f = \mathrm{carrying}(u, c) \vee f = \mathrm{loc}(c)) \rightarrow \\
&\qquad\qquad \mathit{Occlude}(t, f, p)]
\end{aligned} \tag{7}$$

To be able to conveniently prove persistency over qualitative time intervals we would like to express these timepoint occlusion formulas using interval occlusion. The definition of interval occlusion in terms of timepoint occlusion, repeated here with the additional argument for reified action occurrences, is:

$$\forall t_1, t_2, f, p\,[\mathit{Occlude}(t_1, t_2, f, p) \leftrightarrow \exists t\,[t_1 < t \leq t_2 \wedge \mathit{Occlude}(t, f, p)]] \tag{8}$$

Before this definition can be applied we need to put the occlusion formulas (6) into a form where they contain the sub-formula $\exists t[t_1 < t \leq t_2 \wedge \mathit{Occlude}(t, f, p)]$. According to (8) this is equivalent to, hence can be replaced by, interval occlusion. The following equivalence, proven in Appendix B.3, serves this purpose:

$$\begin{aligned}
&\forall t\,[\Phi(t) \rightarrow \Psi(t)] \equiv \\
&\forall t_1, t_2\,[\exists t\,[t_1 < t \leq t_2 \wedge \Phi(t)] \rightarrow \exists t\,[t_1 < t \leq t_2 \wedge \Psi(t)]]
\end{aligned} \tag{9}$$

Let the antecedent and consequent of the occlusion implication (6), with t_1 and t_2 renamed to t_3 and t_4, be Φ and Ψ as in:

$$\begin{aligned}
\Phi(t) &\stackrel{\mathrm{def}}{=} t_3 < t \leq t_4 \wedge \mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \wedge P(\overline{y}) \wedge \\
&\qquad (f = f_1(\overline{z}) \vee \cdots \vee f = f_n(\overline{z})) \\
\Psi(t) &\stackrel{\mathrm{def}}{=} \mathit{Occlude}(t, f, p)
\end{aligned}$$

Then the occlusion formula (6) is rewritten according to theorem (9) as:

$$\begin{aligned}
&\forall t_1, t_2, t_3, t_4, f, \overline{w}, \overline{x}, p\,[\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4 \wedge \\
&\qquad\qquad \mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \wedge P(\overline{y}) \wedge \\
&\qquad\qquad (f = f_1(\overline{z}) \vee \cdots \vee f = f_n(\overline{z}))] \rightarrow \\
&\qquad\qquad\qquad \exists t\,[t_1 < t \leq t_2 \wedge \mathit{Occlude}(t, f, p)]]
\end{aligned}$$

The right hand side of the implication is now in the form required for the application of definition (8). Replacing it and reducing the scope of the existential quantifier on the left hand side results in:

$$\begin{aligned}
&\forall t_1, t_2, t_3, t_4, f, \overline{w}, \overline{x}, p\,[\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4] \wedge \\
&\qquad\qquad \mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \wedge P(\overline{y}) \wedge \\
&\qquad\qquad (f = f_1(\overline{z}) \vee \cdots \vee f = f_n(\overline{z})) \rightarrow \\
&\qquad\qquad\qquad \mathit{Occlude}(t_1, t_2, f, p)]
\end{aligned}$$

The sub-formula $\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4]$ expresses the existence of a shared point between the intervals $(t_1, t_2]$ and $(t_3, t_4]$. One can construct the corresponding interval algebra relation $\{o, s, d, f, =, fi, di, si, oi\}$ by collecting all primitive interval relations that express some form of overlapping. Using the end-point comparison tree described by Thornton et. al. [38] this seemingly complex relation can be simplified to the end-point comparison $t_2 > t_3 \wedge t_1 < t_4$. To point out the correspondence between the existentially quantified timepoint sub-formula and the end-point comparison definition of interval overlap, proved in Appendix B.4, we will make use of a new predicate *Overlap*, defined as follows:

$$\forall t_1, t_2, t_3, t_4\,[t_2 > t_3 \wedge t_1 < t_4 \leftrightarrow \text{Overlap}(t_1, t_2, t_3, t_4)]$$

Introducing the *Overlap* predicate into the occlusion formula puts all occlusion formulas into the general form:

$$\begin{aligned}\forall t_1, t_2, t_3, t_4, f, \overline{w}, \overline{x}, p\,[&\text{Overlap}(t_1, t_2, t_3, t_4) \wedge \\ &\mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \wedge P(\overline{y}) \wedge \\ &(f = f_1(\overline{z}) \vee \cdots \vee f = f_n(\overline{z})) \rightarrow \\ &\mathit{Occlude}(t_1, t_2, f, p)]\end{aligned} \tag{10}$$

The specific form of the example *attach* occlusion formula is:

$$\begin{aligned}\forall t_1, t_2, t_3, t_4, f, u, c, l, p\,[&\text{Overlap}(t_1, t_2, t_3, t_4) \wedge \\ &\mathit{Occurs}(t_3, t_4, \text{attach}(u, c), p) \wedge \\ &\mathit{Holds}(t_3, \text{loc}(c), l, p) \wedge \\ &\forall t\,[t_3 \leq t \leq t_4 \rightarrow \mathit{Holds}(t, \text{loc}(u), l, p)] \wedge \\ &(f = \text{carrying}(u, c) \vee f = \text{loc}(c)) \rightarrow \\ &\mathit{Occlude}(t_1, t_2, f, p)]\end{aligned} \tag{11}$$

To complete the transformation to interval occlusion, and take advantage of the possibility to prove a fluent persistent over an entire temporal interval, the persistence formulas must be replaced by formulas of the same form as formula (3) in Section 3.2 but including the action occurrence argument:

$$\begin{aligned}\forall t_1, t_2, \overline{x}, p\,[&t_1 < t_2 \wedge \neg \mathit{Occlude}(t_1, t_2, f(\overline{x}), p) \rightarrow \\ &\forall v\,[\mathit{Holds}(t_1, f(\overline{x}), v, p) \leftrightarrow \mathit{Holds}(t_2, f(\overline{x}), v, p)]]\end{aligned} \tag{12}$$

4.4 Transformation to Horn Form

The third translation step rewrites formulas into Horn clause form. The reassignment part of action specifications (5) is now of the general form:

$$t_1 < t_2 \wedge \mathit{Occurs} \wedge P_1 \wedge \cdots \wedge P_m \rightarrow E'_1 \wedge \cdots \wedge E'_n$$

where, as before, each P_i is a condition and each E'_j is a reassignment without the occlusion part. Each such formula is expanded into the implications:

$$\begin{aligned} &t_1 < t_2 \wedge \mathit{Occurs} \wedge P_1 \wedge \cdots \wedge P_m \rightarrow E'_1 \\ &\vdots \\ &t_1 < t_2 \wedge \mathit{Occurs} \wedge P_1 \wedge \cdots \wedge P_m \rightarrow E'_n \end{aligned} \tag{13}$$

This is in Horn form except for those conditions P_i that were specified to hold over the whole action occurrence interval $[t_1, t_2]$, which are still of the complex form:

$$\forall t\, [t_1 \leq t \leq t_2 \rightarrow \mathit{Holds}(t, f(\overline{y}), v, p)]$$

For persistent fluents this is equivalent to $f(\overline{y})$ holding the value v at the first timepoint t_1 and, by virtue of not being occluded at any timepoint during the interval, the value v persisting until timepoint t_2:

$$\mathit{Holds}(t_1, f(\overline{y}), v, p) \wedge \neg\exists t\, [t_1 < t \leq t_2 \wedge \mathit{Occlude}(t, f(\overline{y}), p)]$$

According to the definition of interval occlusion (8) this is simply:

$$\mathit{Holds}(t_1, f(\overline{y}), v, p) \wedge \neg \mathit{Occlude}(t_1, t_2, f(\overline{y}), p) \tag{14}$$

The occlusion formulas (10) are in Horn form except for the disjunction over fluents. Rewriting the formula by distributing conjunction over the fluent disjunction, and then distributing the resulting conjunction over the implication, results in one implication for each equality $f = f_i(\overline{z})$. Applying equivalence (4) in the left direction to each implication results in one occlusion definition for each fluent:

$$\begin{aligned} &\forall t_1, t_2, t_3, t_4, \overline{w}, \overline{x}, p\, [\mathrm{Overlap}(t_1, t_2, t_3, t_4) \wedge \mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \wedge P(\overline{y}) \rightarrow \\ &\qquad\qquad \mathit{Occlude}(t_1, t_2, f_1(\overline{z}), p)] \\ &\vdots \\ &\forall t_1, t_2, t_3, t_4, \overline{w}, \overline{x}, p\, [\mathrm{Overlap}(t_1, t_2, t_3, t_4) \wedge \mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \wedge P(\overline{y}) \rightarrow \\ &\qquad\qquad \mathit{Occlude}(t_1, t_2, f_n(\overline{z}), p)] \end{aligned}$$

Each of these formulas contain the action conditions $P(\overline{y})$. This reflects the fact that attempting to execute an action when its conditions are not satisfied will have no effect. We will perform an additional simplification on the occlusion formulas by removing these conditions. In our application, the synthesis and reasoning about plans and actions to execute in the future, considering actions that are not executable is not really relevant. When only considering actions

that can actually be executed this simplification has no effect on the results. Each simplified occlusion formula is then of the form:

$$\forall t_1, t_2, t_3, t_4, \overline{w}, \overline{x}, p\,[\text{Overlap}(t_1, t_2, t_3, t_4) \wedge \mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \rightarrow \mathit{Occlude}(t_1, t_2, f(\overline{z}), p)] \quad (15)$$

Returning to the example logistics narrative fragment, the occlusion formula (11) is instantiated over the fluents *carrying* and *loc*, and simplified by ignoring the conditions for executing the action:

$$\forall t_1, t_2, t_3, t_4, u, c, p\,[\text{Overlap}(t_1, t_2, t_3, t_4) \wedge \mathit{Occurs}(t_3, t_4, \text{attach}(u, c), p) \rightarrow \mathit{Occlude}(t_1, t_2, \text{carrying}(u, c), p)]$$

$$\forall t_1, t_2, t_3, t_4, u, c, p\,[\text{Overlap}(t_1, t_2, t_3, t_4) \wedge \mathit{Occurs}(t_3, t_4, \text{attach}(u, c), p) \rightarrow \mathit{Occlude}(t_1, t_2, \text{loc}(c), p)]$$

To complete the transformation to Horn clause form we need to deal with the equivalence in persistence statements (12) and the *Occurs* definition (3). Both can be rewritten in the form of two implications, however, in each case only one direction is of interest. Persistence will only be proved forward in time, and occurrences are only proved from the occurrence set argument. These implications are then in Horn form:

$$\forall t_1, t_2, \overline{x}, v, p\,[t_1 < t_2 \wedge \neg \mathit{Occlude}(t_1, t_2, f(\overline{x}), p) \wedge \mathit{Holds}(t_1, f(\overline{x}), v, p) \rightarrow \mathit{Holds}(t_2, f(\overline{x}), v, p)] \quad (16)$$

$$\forall t_1, t_2, a, p\,[\mathit{Member}(\mathit{occ}(t_1, t_2, a), p) \rightarrow \mathit{Occurs}(t_1, t_2, a, p)] \quad (17)$$

4.5 Circumscription Policy

Before moving on to the encoding of the formulas as logic programming clauses we need to clarify the relation between the current state of the formulas and the TAL circumscription policy, which was defined in Section 2.5. It is designed to capture the commonsense intuition that change does not spontaneously happen unless there is some reason for it. E.g. if the UAV does not fly anywhere then, by default, its location is unchanged. More formally, given a suitable TAL narrative describing the workings of the UAV, we would like to conclude that the value of the UAV's location fluent will persist over a time interval, unless we in fact know that the UAV was involved in some flying activity during the interval.

In practice, restrictions on the form of TAL narratives ensure that the circumscription policy is reducible to predicate completion, even in the general

case without our additional restrictions added to ensure a proper translation into constraint logic programs. In the compilation above, the occlusion formulas are already in a form amenable to completion by replacing the implications by equivalences, which is the standard way of circumscribing *Occlude* and *Occurs* in the first-order version of TAL narratives. In the case of our logic program encoding, predicate completion can be viewed as the declarative semantics of the negation as failure mechanism of logic programs. This fact suggests that we should leave the occlusion formulas as they are and rely on the "built-in" completion of Prolog to implement the circumscription policy.

However, the situation is complicated when one considers planning. There is no action sequence given beforehand that determines occlusion once and for all. On the contrary, the occlusion concept is continually used to prove fluent value persistence while, at the same time, actions are added to the reified action occurrence set that might affect and modify the very same occlusion. Consequently, we need to continually ensure that considered actions do not violate any persistence proofs that have already been assumed. The Constraint Handling Rules (CHR) framework [10] together with the use of temporal constraint formalisms make a flexible solution to this problem possible. By expressing occlusion using constraint handling rules instead of Prolog clauses we ensure that the conditions are automatically reevaluated whenever needed. The next section presents these rules and explains how they implement a dynamic action/persistence conflict resolution.

Finally, action occurrences should also be circumscribed. Since occlusion will be represented by constraint handling rules, and occlusion depends on action occurrences, we will need to express action occurrences as constraints too. Every time an action is added to the action occurrence set, passed around in the p argument, we will also add it to the constraint store. While the action occurrence argument represents a possibly incompletely specified action sequence, the constraint store is implicitly closed. Therefore the collection of action occurrences added to the store serves as a representation of the circumscription of *Occurs*.

4.6 Clauses and Rules

The fourth and final translation step encodes the TAL formulas as Prolog clauses, constraint handling rules, and finite domain constraints. We start with the *Occurs* definition (17):

```
occurs(T1,T2,A,P) :-
      member(occ(T1,T2,A),P), circ_occurs(T1,T2,A).
```

```
member(A,[A|Pp]).
member(A,P) :-
       nonvar(P), P = [A1|P1], A \== A1, member(A,P1).
```

This deviates from a literal translation into Prolog in two ways. First, the `circ_occurs` constraint was added to make sure the action occurrence is present in the constraint store, where it can interact with the constraint handling rules for occlusion, as noted in Section 4.5. Second, the `member` predicate uses Prolog's built in list syntax rather than the *cons* function, introduced in Section 3.1. Though our implementation of `member` is not the same as the standard one provided by the Prolog list library. The reason is that if one adds an action, and is subsequently forced to backtrack due to reaching a dead end in the search space, then the library version supplies an infinite number of irrelevant backtracking possibilities. Instead of trying some other action, one would be stuck considering syntactically alternative ways of adding the same action to the list. Action occurrence terms should be collected in a *set*, without consideration of their syntactic order. This is the behaviour of the fluent composition function $\circ$, used to represent world states in the Fluent Calculus [37]. Our implementation of `member` borrows Thielscher's fluent composition predicate `holds` from the Fluent Calculus FLUX system [36], which provides only relevant backtracking opportunities.

Next, each of the n action reassignment formulas in (13) is of the form:

$$\forall t_1, t_2, \overline{w}, \overline{x}, p\,[t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, a(\overline{w}), p) \wedge \\ \mathit{Holds}(t_1, f_1(\overline{y_1}), v_1, p) \wedge \cdots \wedge \mathit{Holds}(t_1, f_m(\overline{y_m}), v_m, p) \rightarrow \\ \mathit{Holds}(t_2, f(\overline{z}), v, p)]$$

with, according to (14), a $\neg \mathit{Occlude}(t_1, t_2, f_i(\overline{y}), p)$ for those conditions that should persist over the entire action occurrence interval. This is written as a Prolog clause:

```
holds(T2,f(z),v,P) :-
       types(x), T1 #< T2, occurs(T1,T2,a(w),P),
       holds(T1,f1(y1),v1,P), ···, holds(T1,fm(ym),vm,P).
```

with the addition of `not_occlude(T1,T2,`$f_i(\overline{y})$`,P)` for persisting conditions. Since the timepoints t_1 and t_2 are usually not instantiated as integers, the inequality between them can not be expressed by Prolog's built-in arithmetic operator. Instead the inequality is added to the finite domain constraint solver store using `#<`. Finally, $\mathcal{L}$(FL) is an order-sorted logic that may associate type restrictions on fluent arguments. Prolog is not typed, but the *types* predicate is expanded into unary type clauses that ensure that action argument variables only take on values from the value domain that corresponds to their type. For

example, the action of attaching crates involves a variable c ranging over the set of crates. The Prolog translation of its action reassignment formula would include a unary predicate `crate(C)`, and the set of crates that the variable ranges over would be added to the program as assertions:

```
crate(crate1).
crate(crate2).
crate(crate3).
```

A special `timepoint` type predicate ensure that the timepoint variables `T1` and `T2` are assigned a finite domain, as required by the finite domain constraint solver.

Before moving on, note that Prolog does not have classical negation. Consequently negative effects, such as $\neg \mathit{Holds}(t_2, \mathrm{loc}(c), l, p)$ in our attach example, can not be encoded explicitly in the logic program. Negation is implicit in the assumption of complete knowledge and negation as failure. Thus, if a fluent has not been explicitly mentioned in the effects of some action, it can be proven false using negation as failure. One might wonder whether persistence would not allow propagating a fluent's value from some earlier timepoint, even in the presence of an intermediate action with the negation of the fluent among its effects, since that intermediary negative effect is not explicitly encoded. However, the fluent is still explicitly *occluded*, as specified by the occlusion part of the action specification. Any action that occludes the fluent while overlapping the persistence interval will, by the definition of interval occlusion, also occlude the persistence interval, and persistence will not be applicable.

Once again returning to the logistics example, the negative effect of *attach* in formula (7) is accordingly left out of the logic program while the positive effect, which is of the form:

$$\begin{aligned}\forall t_1, t_2, u, c, l, p\, [t_1 < t_2 \wedge \mathit{Occurs}(t_1, t_2, \mathrm{attach}(u, c), p) \wedge \\ \mathit{Holds}(t_1, \mathrm{loc}(c), l, p) \wedge \mathit{Holds}(t_1, \mathrm{loc}(u), l, p) \wedge \\ \neg \mathit{Occlude}(t_1, t_2, \mathrm{loc}(u), p) \rightarrow \\ \mathit{Holds}(t_2, \mathrm{carrying}(u, c), \mathrm{true}, p)]\end{aligned}$$

is compiled into:

```
holds(T2,carrying(U,C),true,P) :-
       uav(U), crate(C), timepoint(T1), timepoint(T2),
       T1 #< T2, occurs(T1,T2,attach(U,C),P),
       holds(T1,loc(C),L,P), holds(T1,loc(U),L,P),
       not_occlude(T1,T2,loc(U),P).
```

where `uav/1` and `crate/1` are the type predicates that prevent the action from being applied to anything else than UAVs and crates, and `timepoint/1` assign finite domains to the timepoint variables.

According to the declarative CHR semantics (page 7, [10]), each occlusion formula of the form (15), repeated here:

$$\forall t_1, t_2, t_3, t_4, \overline{w}, \overline{x}, p\,[\text{Overlap}(t_1, t_2, t_3, t_4) \wedge \mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \rightarrow \mathit{Occlude}(t_1, t_2, f(\overline{z}), p)]$$

is equivalent to a constraint handling rule of the following form:

`overlap(T1,T2,T3,T4), occurs(T3,T4,`$a(\overline{w})$`,P) ==>`
`  occlude(T1,T2,`$f(\overline{x})$`,P).`

The purpose of this rule is to occlude fluents that are affected by action occurrences in the action occurrence set p. While planning, however, the set p is intentionally incompletely specified to allow the addition of new actions. Whether or not some specific action is present in the current action occurrence set p may not be known, and supplying the corresponding Prolog goal `occurs(T3,T4,`$a(\overline{w})$`,P)` would actually *add* the action if not already present, since this is a valid way of proving that goal. The same problem applies to the occlusion constraint on the right hand side of the rule since it also depends on the actions occurrence set. The best we can do is to make sure that fluents affected by actions that we have assumed occur, up to the current point in the planning process, are occluded. Those actions are stored using the `circ_occurs` constraint, as noted in the translation of the *Occurs* definition (17). By replacing the use of predicates that depend on the incompletely specified set `P` by the corresponding *constraints*, `circ_occurs` and `occlude`, we take all actions that are currently in the plan into consideration. The CHR framework then assumes responsibility for the rest of the work by continually monitoring these constraints for updates. Whenever the action occurrence set `P` is modified, a new `circ_occurs` constraint is added, and the occlusion rule is automatically reevaluated. The new constraint handling rule is then:

`overlap(T1,T2,T3,T4), circ_occurs(T3,T4,`$a(\overline{w})$`) ==>` (18)
`  occlude(T1,T2,`$f(\overline{x})$`).`

The persistence formulas are of the form (16), repeated here:

$$\forall t_1, t_2, \overline{x}, v, p\,[t_1 < t_2 \wedge \neg \mathit{Occlude}(t_1, t_2, f(\overline{x}), p) \wedge \mathit{Holds}(t_1, f(\overline{x}), v, p) \rightarrow \mathit{Holds}(t_2, f(\overline{x}), v, p)]$$

and are encoded as prolog clauses:

```
holds(T2,f(x̄),v,P) :-                                              (19)
      timepoint(T1), timepoint(T2), T1 #< T2,
      not_occlude(T1,T2,f(x̄)), holds(T1,f(x̄),v,P).
```

The negated *Occlude* predicate is encoded as a constraint `not_occlude` to allow the proper interaction with the `occlude` constraint in the occlusion CHR rule (18). As before, the best we can do is to consider all decisions made up to the current point in the planning process. CHR will then remember to reevaluate the rules whenever new descisions are made. Specifically, when the persistence formula (19) is used to propagate a fluent's value, the `not_occlude` constraint would be added to the constraint store. If an overlapping action interval that affects the fluent's value was already present in the store, the fluent would not really be persistent. The occlusion rule (18) should be reevaluated and detect the conflict, adding a contradictory `occlude` constraint to the store. The only missing piece is a rule to force backtracking when such conflicts, caused by a persistence assumptions and action occurrences, are detected:

```
not_occlude(T1,T2,f(x̄)), occlude(T1,T2,f(x̄)) ==> fail.
```

However, there is one problem with this initial encoding attempt and it is related to the `overlap` constraint. The CHR rule (18) is a syntactic matching rule. It will only trigger if the constraints on its left hand side are explicitly represented in the constraint store. Action occurrences are stored explicitly in the `circ_occurs` constraint, but the same is not true for the `overlap` constraint. Unless we have complete knowledge about the ordering of actions and represent this knowledge as explicit constraints, the rule may not trigger to detect conflicts between persistence assumptions and actions that are added to the plan. But we would like to have both a compact and *incomplete* representation of temporal constraints. This would allow the implementation to follow a strategy of least commitment concerning the orderings of actions and to generate partially ordered plans.

Solving this problem involves restructuring the occlusion formulas so that knowledge of interval overlap is not a *prerequisite* to conflict detection, but a *consequence*. The occlude formulas (15) have their relations rearranged into the logically equivalent form:

$$\forall t_1, t_2, t_3, t_4, \overline{w}, \overline{x}, p\,[\neg \mathit{Occlude}(t_1, t_2, f(\overline{z}), p) \wedge \mathit{Occurs}(t_3, t_4, a(\overline{w}), p) \rightarrow \neg\mathrm{Overlap}(t_1, t_2, t_3, t_4)]$$

Likewise, the new constraint handling rule, replacing (18), is:

```
not_occlude(T1,T2,f(x̄)), circ_occurs(T3,T4,a(w̄)) ==>
      not_overlap(T1,T2,T3,T4).
```

Both the `not_occlude` and `circ_occurs` constraints have complete representations in the constraint store, thereby ensuring that the rule will trigger whenever these constraints are modified in the store. Thus, when clause (19) has been used to prove a fluent persistent over an interval $(t_1, t_2]$, adding a `not_occlude` constraint in the process, and an action that threatens to modify the fluent occurs during interval $(t_3, t_4]$, then the rule will act to constrain the two intervals not to overlap. The `not_overlap` predicate adds the corresponding interval algebra end-point constraints to the finite domain constraint solver discussed in Section 3.2.1. If the addition of these temporal constraints results in inconsistency, Prolog will backtrack, forcing the choice of another action or the removal of the persistence assumption. In other words, the mechanism resolves threats posed by actions, which could potentially disturb fluents that are required to be persistent, by introducing restrictions on the order of actions and persistence intervals. Backtracking is only necessary when further ordering restrictions are not possible.

In the logistics example the resulting persistence clauses and occlusion rules are as follows:

```
holds(T2,carrying(U,C),V,P) :-
  timepoint(T1), timepoint(T2), T1 #< T2,
  not_occlude(T1,T2,carrying(U,C)),
  holds(T1,carrying(U,C),V,P).
holds(T2,loc(C),V,P) :-
  timepoint(T1), timepoint(T2), T1 #< T2,
  not_occlude(T1,T2,loc(C)),
  holds(T1,loc(C),V,P).
not_occlude(T1,T2,carrying(U,C)),
  circ_occurs(T3,T4,attach(U,C)) ==>
  not_overlap(T1,T2,T3,T4).
not_occlude(T1,T2,loc(C)),
  circ_occurs(T3,T4,attach(U,C)) ==>
  not_overlap(T1,T2,T3,T4).
```

Finally, observation statements are easily encoded as:

`holds(0,`f`,`v`,P).`

In the case of the UAV example the encoding is:

```
holds(0,loc(uav1),pos(300,90),P).
holds(0,loc(crate1),pos(300,90),P).
```

The compilation results in executable constraint logic programs for deductive planning and reasoning. An example program is integrated with the UAV system in Chapter 5 and listed in full in the Appendix.

4.7 Summary

Although many translation steps are equivalence preserving, some are necessarily not, or else we would not have arrived at a logic program with deductive planning capability. We summarize the points of difference here.

- An obvious restriction on the original TAL is the form of $\mathcal{L}$(ND) input theories, which limits the applicability of the compilation to domains that can be adequately described while satisfying these constraints. The translation from $\mathcal{L}$(ND) to the first-order $\mathcal{L}$(FL) is unmodified. Instead, additional translations are subsequently performed.
- Timepoint occlusion is replaced by interval occlusion in the action specifications. Interval occlusion is defined in terms of TAL's original timepoint occlusion by (8), and the modification is expressed in terms of an equivalence preserving rewrite (Appendix B.3).
- Interval overlap is expressed succinctly using an *Overlap* predicate that is equivalent to the formula expressing interval overlap that resulted from the translation to interval occlusion (Appendix B.4).
- A formula (12) that allows the use of fluent persistency over intervals of any length is added. It is entailed by the definition of interval occlusion and persistence (Appendix B.1, Appendix B.2), but is weaker than the interval persistence formula (2) in Section 3.2 since it does not (by itself) constrain the fluent's value *during* the interval. The stronger formula could presumably be added, if needed, but we have not investigated this further.
- The formulas are expanded into equivalent formulas in Horn form, which is possible due to the restrictions on the original $\mathcal{L}$(FL) narrative. An additional requirement is that fluents that are specified to hold a specific value throughout the action occurrence interval are persistent. Otherwise one would need to use universal quantification over timepoints in that interval.
- Occlusion formulas are significantly simplified by removing the action preconditions (which are independently present in action effect formulas).

This weakens the theory since actions may occlude fluents, preventing application of persistence, even though the actions may in fact have had no effects due to their preconditions not being satisfied. Though, when planning, such cases will never actually appear since it would be pointless to add actions without effects to the narrative.

- When the persistence equivalence (12) is turned into a Horn form implication, this weakens the theory to persistence forward in time. We have not investigated whether the other direction could be included to allow *postdictive* conclusions about fluent values at earlier timepoints based on knowledge about later timepoints.

- CHR rules minimize occlusion and occurrences in the same way the circumscription policy would have done, from a theoretical point of view. However, the difference is that the proof *mechanism* is applicable even to a dynamically changing narrative, which is the case in deductive planning.

- The conversion of the Horn formulas into Prolog clauses strengthens the theory in the sense that anything that was previously unknown is now false. But it limits the applicability of our methods, as noted below.

- Finally, the use of a finite domain constraint solver for (in)equalities weakens the theory since we can not draw conclusions e.g. about fluent values outside the bounded time line. In practice this problem can be circumvented by adjusting the size of the bound.

The generated constraint logic programs are thus associated with severe restrictions on the form of TAL theories that can be used as input. Specifically, the requirement of a completely specified initial state and the negation-as-failure semantics of Prolog make the compilation, as it is described above, inapplicable to planning problems that involve incomplete information, and the restrictions imposed by Horn-clause form limit syntactic expressivity and the use of quantifiers.

It is important to note, however, that without some mechanism for incremental minimization of *Occlude* and *Occurs*, deductive planning would be impossible. Previous work with TAL (e.g. the most recent language specification [8] and the most used reasoning tool [18]) has assumed the provision of completely specified narratives, detailing what actions occur when, *before* any reasoning process commences. The constraint handling rules, introduced above, take care of the minimization in an automated fashion and allow the dynamic generation of new narratives. They significantly simplify the proof process and make automated planning efficient. Building on this basis will

require augmentation of the basic Prolog inference mechanism through a meta-interpreter, or replacing Prolog with a more powerful theorem prover, but such efforts should ultimately benefit from the inherent capability of the underlying TAL language to compactly represent incompletely specified information, and make possible a more liberal use of quantifiers.

Chapter 5

Composite Actions

The previous chapters have considered plans in the form of sets of primitive action occurrences that are partially ordered on a time line. But such a view is not sufficiently general to capture the entire range of plans that we would intuitively expect an intelligent agent to be able to use. Specifically, there is no concept of conditional action occurrences or of repetition of action occurrences. If such constructs are introduced they form a basic scripting language that can be used to express more complex instructions for the agent to carry out.

This chapter extends the high-level $\mathcal{L}$(ND) lanugage syntax to cover sequences of action occurrences, conditional occurrences, nondeterministic choice actions, and loops. A semantics for these *composite actions* is provided by a corresponding extension of the *Trans* function that translates the high-level language into the first-order language $\mathcal{L}$(FL). The result, however, is no longer a first-order language. A move to fixpoint logic is necessary since we make use of the least fixpoint operator to capture the semantics of loop constructs. Finally, a compilation of composite actions into Prolog clauses is described, which permits the execution of complex actions in the logic programming framework that was developed in Chapter 4.

5.1 Syntax and Semantics

The syntax and semantics of composite actions are defined by extending the translation function *Trans* defined in Section 2.3.1. The following additions illustrate both the $\mathcal{L}$(ND) syntax of composite actions and the result of translating them to $\mathcal{L}$(FL) with the least fixpoint operator. Let a be a primitive action, C a formula expressing some condition, and A a composite action. We

extend *Trans* with sequences, conditional action occurrences, non-deterministic choice, and loops as follows:

$$\begin{array}{l} \mathit{Trans}([t_1,t_2]\ A_1;A_2) \stackrel{\text{def}}{=} \\ \quad \exists t[t_1 < t < t_2 \wedge \mathit{Trans}([t_1,t]\ A_1) \wedge \mathit{Trans}([t,t_2]\ A_2)] \\ \mathit{Trans}([t_1,t_2]\ \mathit{if}\ C\ \mathit{then}\ A) \stackrel{\text{def}}{=} \\ \quad \mathit{Trans}([t_1]\ C) \rightarrow \mathit{Trans}([t_1,t_2]\ A) \\ \mathit{Trans}([t_1,t_2]\ \mathit{choose}\ x\ A) \stackrel{\text{def}}{=} \\ \quad \exists x[\mathit{Trans}([t_1,t_2]\ A)] \\ \mathit{Trans}([t_1,t_2]\ \mathit{do}\ A\ \mathit{until}\ C) \stackrel{\text{def}}{=} \\ \quad \mu X(x,y)[\mathit{Trans}([x]\ \neg C) \rightarrow \exists z[x < z \leq y \wedge \mathit{Trans}([x,z]\ A) \wedge X(z,y)]](t_1,t_2) \end{array}$$

5.2 Fixpoint Expansion

Although the semantics of the first three composite actions is relatively self explanatory, it might not be immediately obvious what the fixpoint formula in the fourth case means. A least fixpoint formula is of the general form:

$$\mu X(\overline{x})[\Phi](\overline{t})$$

where X is k-ary relational variable, $\overline{x}$ is k variables, and Φ is a formula with only positive occurrences of X. Variables in $\overline{x}$ that are unbound in Φ serve as arguments for the fixpoint function and their values are provided in $\overline{t}$. The semantics of the expression can be defined model-theoretically as a least fixpoint relation on sets. However, we will instead illustrate its meaning using the following syntactic expansion theorem:

$$\mu X(\overline{x})[\Phi](\overline{t}) \equiv [\bigvee_{i \in \mathbb{Z}} X^i](\overline{t}) \quad \text{where} \quad X^i \equiv \begin{cases} \mathit{false} & \text{if } i = 0 \\ \Phi[X(\overline{x}) \leftarrow X^{i-1}(\overline{x})] & \text{otherwise} \end{cases}$$

The fixpoint formula can thus be rewritten as an (infinite) disjunction where each disjunct is an expansion of the formula Φ with occurrences of X replaced by the previous expansion step.

5.2.1 Expansion Example

Consider a UAV logistics task where there is a need for a certain number of crates at a specific location. We would like to instruct the UAV to start delivering crates at timepoint s and stop at some later time t when a sufficient number of crates are available at the target location. For the purpose of illustration we help keep the formulas relatively lucid by abridging the delivery

of a crate as an action *deliver* and the condition to be satisfied as a boolean fluent *sufficient*. A composite action to accomplish the given task would then be written as:

$$[s,t]\ \mathit{do}\ \text{deliver}\ \mathit{until}\ \text{sufficient} \mathrel{\hat{=}} \mathit{true}$$

According to the extended *Trans* function defined above, this macro is expanded into the fixpoint $\mathcal{L}(\text{FL})$ formula:

$$\mu X(x,y)[\neg \mathit{Holds}(x, \text{sufficient}, \mathit{true}) \rightarrow \\ \exists z\,[x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge X(z,y)]](s,t)$$

The first step of the expansion, X^0, is simply *false*. The next step, X^1, is obtained by replacing the occurrence of X in the body of the fixpoint formula with X^0 (which is *false*). The result is rewritten in disjunctive form to help illustrate a uniform structure that will become clear after a couple of expansion steps:

$$\neg \mathit{Holds}(x, \text{sufficient}, \mathit{true}) \rightarrow \\ \exists z\,[x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge X^0(z,y)] \equiv$$

$$\mathit{Holds}(x, \text{sufficient}, \mathit{true}) \vee \\ \exists z\,[x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver})]$$

Again, replacing X by X^1 in the fixpoint formula body and rewriting results in X^2:

$$\neg \mathit{Holds}(x, \text{sufficient}, \mathit{true}) \rightarrow \\ \exists z\,[x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge X^1(z,y)] \equiv$$

$$\mathit{Holds}(x, \text{sufficient}, \mathit{true}) \vee \\ \exists z\,[x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge \\ \quad (\mathit{Holds}(z, \text{sufficient}, \mathit{true}) \vee \exists z_2\,[z < z_2 \leq y \wedge \mathit{Occurs}(z, z_2, \text{deliver})])] \equiv$$

$$\mathit{Holds}(x, \text{sufficient}, \mathit{true}) \vee \\ \exists z\,[x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge \mathit{Holds}(z, \text{sufficient}, \mathit{true})] \vee \\ \exists z, z_2\,[x < z < z_2 \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge \mathit{Occurs}(z, z_2, \text{deliver})]$$

The structure of the formula becomes more clear when completing the third step, replacing X by X^2 and rewriting to get X^3:

$$\neg \mathit{Holds}(x, \text{sufficient}, \mathit{true}) \rightarrow \\ \exists z\,[x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge X^2(z,y)] \equiv$$

$$
\begin{array}{l}
\mathit{Holds}(x, \text{sufficient}, \mathit{true}) \vee \\
\exists z\, [x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge \\
\quad (\mathit{Holds}(z, \text{sufficient}, \mathit{true}) \vee \\
\quad\ \exists z_2\, [z < z_2 \leq y \wedge \mathit{Occurs}(z, z_2, \text{deliver}) \wedge \mathit{Holds}(z_2, \text{sufficient}, \mathit{true})] \vee \\
\quad\ \exists z_2, z_3\, [z < z_2 < z_3 \leq y \wedge \\
\qquad\qquad \mathit{Occurs}(z, z_2, \text{deliver}) \wedge \mathit{Occurs}(z_2, z_3, \text{deliver})])] \equiv \\
\mathit{Holds}(x, \text{sufficient}, \mathit{true}) \vee \\
\exists z\, [x < z \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge \mathit{Holds}(z, \text{sufficient}, \mathit{true})] \vee \\
\exists z, z_2\, [x < z < z_2 \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge \\
\qquad \mathit{Occurs}(z, z_2, \text{deliver}) \wedge \mathit{Holds}(z_2, \text{sufficient}, \mathit{true})] \vee \\
\exists z, z_2, z_3\, [x < z < z_2 < z_3 \leq y \wedge \mathit{Occurs}(x, z, \text{deliver}) \wedge \\
\qquad\qquad \mathit{Occurs}(z, z_2, \text{deliver}) \wedge \mathit{Occurs}(z_2, z_3, \text{deliver})]
\end{array}
$$

The expansion continues indefinitely but the general form of the resulting disjunction is clear:

$$
\begin{array}{l}
\mathit{Holds}(x, \text{sufficient}, \mathit{true}) \vee \\
\exists z\, [x < z \leq y \wedge \\
\quad \mathit{Occurs}(x, z, \text{deliver}) \wedge \\
\quad \mathit{Holds}(z, \text{sufficient}, \mathit{true})] \vee \\
\exists z, z_2\, [x < z < z_2 \leq y \wedge \\
\qquad \mathit{Occurs}(x, z, \text{deliver}) \wedge \\
\qquad \mathit{Occurs}(z, z_2, \text{deliver}) \wedge \\
\qquad \mathit{Holds}(z_2, \text{sufficient}, \mathit{true})] \vee \\
\exists z, z_2, z_3\, [x < z < z_2 < z_3 \leq y \wedge \\
\qquad\qquad \mathit{Occurs}(x, z, \text{deliver}) \wedge \\
\qquad\qquad \mathit{Occurs}(z, z_2, \text{deliver}) \wedge \\
\qquad\qquad \mathit{Occurs}(z_2, z_3, \text{deliver}) \wedge \\
\qquad\qquad \mathit{Holds}(z_3, \text{sufficient}, \mathit{true})] \vee \\
\exists z, z_2, z_3, z_4\, [x < z < z_2 < z_3 < z_4 \leq y \wedge \\
\qquad\qquad\qquad \mathit{Occurs}(x, z, \text{deliver}) \wedge \\
\qquad\qquad\qquad \mathit{Occurs}(z, z_2, \text{deliver}) \wedge \\
\qquad\qquad\qquad \mathit{Occurs}(z_2, z_3, \text{deliver}) \wedge \\
\qquad\qquad\qquad \mathit{Occurs}(z_3, z_4, \text{deliver}) \wedge \\
\qquad\qquad\qquad \mathit{Holds}(z_4, \text{sufficient}, \mathit{true})] \vee \cdots
\end{array}
$$

The above formula corresponds to the expected behaviour of the composite loop action. Each disjunct represents an additional occurrence of the *deliver* action in the series before the *sufficient* condition is satisfied. The unfolding of the composite loop action (in the context of a narrative that specifies the effects of the action, the initial conditions, and the *sufficient* condition) is given

by the first satisfied disjunct in the formula with the unbound variables x and y instantiated by the s and t arguments.

5.3 Compiling Composite Actions

Some automated reasoning method applicable to composite actions is needed before they become useful in practice. Section 3.3 hinted that direct application of first-order theorem proving techniques does not necessarily result in a practical system. Composite actions are translated into fixpoint logic where first-order proof methods are not even applicable. An alternative automated reasoning method is the compilation into constraint logic programs in Chapter 4, which can be extended to the case of composite actions.

The compilation results in a Prolog goal that when evaluated will produce the actual execution trace of primitive action occurrences where conditionals have been resolved, non-deterministic choices instantiated, and iteration fixpoints reached. As before, these primitive action occurrences are stored in a reified action occurrence arument p. Note that only *primitive* action occurrences are reified while *composite* action occurrences are Prolog clauses. The reified action occurrence argument does not contain composite actions but is introduced for compatibility with the deductive planning framework. Reified composite actions is a more complex topic, as discussed in Section 5.4.

5.3.1 The Compilation Function

The first compilation step, the translation from $\mathcal{L}(\mathrm{ND})$ into fixpoint logic, was introduced above. The second, and final step, is the compilation from fixpoint logic into constraint logic programs. The compilation is most easily defined recursively since a composite action may be composed of other composite actions. The compilation will be associated with a function named *Comp* for the purpose of conveniently expressing this recursion. The base case of the compilation function, the primitive action occurrence, is defined as the corresponding Prolog clause:

$$\mathit{Comp}(\mathit{Occurs}(t_1, t_2, a, p)) \stackrel{\text{def}}{=} \texttt{occurs(T1,T2,}a\texttt{,P)}$$

Sequences are composed of two composite actions in series, with constraints on the shared timepoint:

$Comp(\exists t[t_1 < t < t_2 \wedge Trans([t_1, t]\ A_1) \wedge Trans([t, t_2]\ A_2)]) \stackrel{\text{def}}{=}$
`T1 #< T, T #< T2,` $Comp(Trans([t_1, t]\ A_1))$`,` $Comp(Trans([t, t_2]\ A_2))$

Conditionals consist of two complex parts, a condition C and an action A. The translation of the condition C needs special care. If C evaluates to true, the set of action occurrences should include A, however, we do not want the evaluation of C *itself* to affect the set of action occurrences. This is accomplished by minimizing the action occurrences in the variable p through a function `circ` that removes the unbound tail variable in the corresponding Prolog list variable `P`. The result is a new action occurrence variable `Pp` (a legal Prolog variable name, unlike `P'`) where we have assumed complete knowledge about action occurrences. Using this new, circumscribed, variable when evaluating the condition C prevents the addition of new actions. The implication can then be written using the Prolog disjunction operator `;` as:

$Comp(Trans([t_1]\ C) \rightarrow Trans([t_1, t_2]\ A)) \stackrel{\text{def}}{=}$
`circ(P,Pp),` $Comp(Trans([t_1]\ \neg C))$ `;` $Comp(Trans([t_1, t_2]\ A))$

Nondeterministic choices have an existentially quantified variable that is simply replaced by a Prolog variable in the resulting expression:

$Comp(\exists x[Trans([t_1, t_2]\ A)]) \stackrel{\text{def}}{=}$
$Comp(Trans([t_1, t_2]\ A))$ with occurrences of x replaced by `X`

Loops, with the additional action occurrence argument, are compiled in two steps. First, the implicit least fixpoint function X is made explicit as a Prolog predicate `x` that defines the loop. As with conditional actions, the test C is evaluated with the action occurrence variable `P` replaced by a circumscribed version `Pp`. Secondly, a Prolog goal calls the new predicate to evaluate the actual parameters passed to the function:

$Comp(\mu X(x, y, p)[Trans([x]\ \neg C) \rightarrow \exists z[x < z \leq y \wedge Trans([x, z]\ A) \wedge X(z, y, p)]](t_1, t_2, p)) \stackrel{\text{def}}{=}$

a new Prolog predicate:

`x(X,Y,P) :-` `circ(P,Pp),` $Comp(Trans([x]\ C))$ `;`
`X #< Z, Z #=< Y,` $Comp(Trans([x, z]\ A))$`, x(Z,Y,P).`

and the goal:

`x(`t_1`,`t_2`,P)`

Finally, since Prolog is not typed, nondeterministic choice variables and quantified variables in test conditions require additional clauses that represent unary type predicates. These ensure that the variables can only be bound to objects of the correct type. As before, the special `timepoint` clause assigns a finite domain to the timepoint constraint variables.

5.3.2 An Example Composite Action

Consider the *attach* action TAL narrative from Chapter 4 extended with a converse *drop* action and a *fly* action. These primitive actions can be used to write composite actions for the delivery of crates between locations. E.g., one such logistics task is to fly a UAV to fetch crates at a location *loc1* and deliver them to another location *loc2* until there are at least two crates there. All of the different types of composite actions are present in its specification:

$$
\begin{array}{l}
[0,t]\ \mathit{do}\ (\mathit{if}\ \neg\mathrm{loc}(\mathrm{uav1}) \hat{=} \mathrm{loc1}\ \mathit{then}\ \mathrm{fly}(\mathrm{uav1},\mathrm{loc1});\\
\qquad \mathit{choose}\ c\ (\mathrm{attach}(\mathrm{uav1},c);\\
\qquad\qquad \mathrm{fly}(\mathrm{uav1},\mathrm{loc2});\\
\qquad\qquad \mathrm{drop}(\mathrm{uav1},c)))\\
\quad \mathit{until}\ \exists c_1,c_2[\mathrm{loc}(c_1) \hat{=} \mathrm{loc2} \wedge \mathrm{loc}(c_2) \hat{=} \mathrm{loc2} \wedge c_1 \neq c_2]
\end{array}
$$

The composite action is first translated into $\mathcal{L}(\mathrm{FL})$:

$$
\begin{array}{l}
\mu X(x,y,p)[\\
\quad \neg\exists c_1,c_2\,[\mathit{Holds}(x,\mathrm{loc}(c_1),\mathrm{loc2},p) \wedge \mathit{Holds}(x,\mathrm{loc}(c_2),\mathrm{loc2},p) \wedge c_1 \neq c_2] \rightarrow\\
\quad \exists z\,[x < z \leq y \wedge\\
\qquad \exists t_1\,[x < t_1 < z \wedge\\
\qquad\quad (\neg \mathit{Holds}(x,\mathrm{loc}(\mathrm{uav1}),\mathrm{loc1},p) \rightarrow\\
\qquad\quad \mathit{Occurs}(x,t_1,\mathrm{fly}(\mathrm{uav1},\mathrm{loc1}),p)) \wedge\\
\qquad\quad \exists c\,[\exists t_2\,[t_1 < t_2 < z \wedge\\
\qquad\qquad\quad \mathit{Occurs}(t_1,t_2,\mathrm{attach}(\mathrm{uav1},c),p) \wedge\\
\qquad\qquad\quad \exists t_3\,[t_2 < t_3 < z \wedge\\
\qquad\qquad\qquad \mathit{Occurs}(t_2,t_3,\mathrm{fly}(\mathrm{uav1},\mathrm{loc2}),p) \wedge\\
\qquad\qquad\qquad \mathit{Occurs}(t_3,z,\mathrm{drop}(\mathrm{uav1},c),p)]]]] \wedge\\
\qquad X(z,y,p)]](0,t,p)
\end{array}
$$

The logic program compilation then creates a new Prolog predicate that defines the least fixpoint loop formula, with additional type and timepoint predicates:

```
x(X,Y,P) :-
  circ(P,Pp1), crate(C1), crate(C2), holds(X,loc(C1),loc2,Pp1),
  holds(X,loc(C2),loc2,Pp1), C1 \== C2 ;
    timepoint(Z), X #< Z, Z #=< Y,
      timepoint(T1), X #< T1, T1 #< Z,
      ((circ(P,Pp2), holds(X,loc(uav1),loc1,Pp2)) ;
       occurs(X,T1,fly(uav1,loc1),P)),
      crate(C), timepoint(T2), timepoint(T3), T1 #< T2, T2 #< Z,
        occurs(T1,T2,attach(uav1,C),P),
        T2 #< T3, T3 #< Z,
```

```
        occurs(T2,T3,fly(uav1,loc2),P),
        occurs(T3,Z,drop(uav1,C),P),
    x(Z,Y,P).
```

Finally, the Prolog goal consists of calling the new predicate with the actual parameters:

```
?- x(0,T,P), timepoint_variables(L), labeling([],L).
```

The `timepoint_variables` predicate collects all the finite domain constraint variables expressing temporal constraints and the `labeling` predicate ensures their consistency (or forces backtracking otherwise).

Evaluating the goal in an initial state where the UAV is already at location `loc1` binds the action occurrence argument `P` to the following sequence of action occurrences:

$occ(t_1, t_2, \text{attach}(\text{uav1}, \text{crate1}))$
$occ(t_2, t_3, \text{fly}(\text{uav1}, \text{loc2}))$
$occ(t_3, t_4, \text{drop}(\text{uav1}, \text{crate1}))$
$occ(t_4, t_5, \text{fly}(\text{uav1}, \text{loc1}))$
$occ(t_5, t_6, \text{attach}(\text{uav1}, \text{crate2}))$
$occ(t_6, t_7, \text{fly}(\text{uav1}, \text{loc2}))$
$occ(t_7, t_8, \text{drop}(\text{uav1}, \text{crate2}))$
$0 < t_1 < t_2 < t_3 < t_4 < t_5 < t_6 < t_7 < t_8 \leq t$

5.4 Reified Composite Actions

The composite actions that we have presented are given by an operator and executed by the robot. This is similar to how primitive actions were used in TAL before our deductive planning extensions. Just as reified action occurrences made deductive planning with primitive action occurrences possible, one would like to use reification or some other scheme to make deductive planning with composite actions possible. However, since the semantics of composite actions are expressed in fixpoint logic there is no complete proof procedure that can be applied in the general case. Moreover, loops and conditionals make up a Turing complete language, thereby equating the generation of composite actions with the general problem of program synthesis. These considerations bodes for both great challenges and great rewards for future research in this direction.

Chapter 6

A UAV Logistics Application

We call the constraint logic programming methodology just described PARADOCS for Planning And Reasoning As DeductiOn with ConstraintS. It has been integrated with our autonomous unmanned aerial vehicle system and applied to solving logistics problems to test the viability of the method.

The integrated system can use the set of actions in the logistics scenario from Section 5.3.2 to distribute crates according to a declarative goal specification by generating a plan that involves repeatedly attaching a crate, flying, and then dropping it at the intended location. The UAV operator uses a graphical drag'n'drop user interface to set up the goal specification for a mission. The goals are sent to a Prolog program that is compiled from the logistics TAL narrative. Executing the program corresponds to proving the goal, and a partial order plan is extracted from the constraint stores that result from the proof. The operator uses the GUI to choose a linearization, which is compiled into a sequence of commands that can be executed by the UAV system. The non-occlusion constraints on fluent values are then monitored during the actual execution to make sure that it proceeds according to the plan. A failure of a monitor condition triggers a plan repair where the current state of execution is used by PARADOCS to insert recovery actions that will put the execution back on track and ensure that the goals are satisfied.

This chapter describes the implementation of the above scenario when coupled to the helicopter simulator. We are currently at work building a winch system with an electro-magnetic hook and trying to solve the vision problems associated with detecting crates and hovering with sufficient precision over

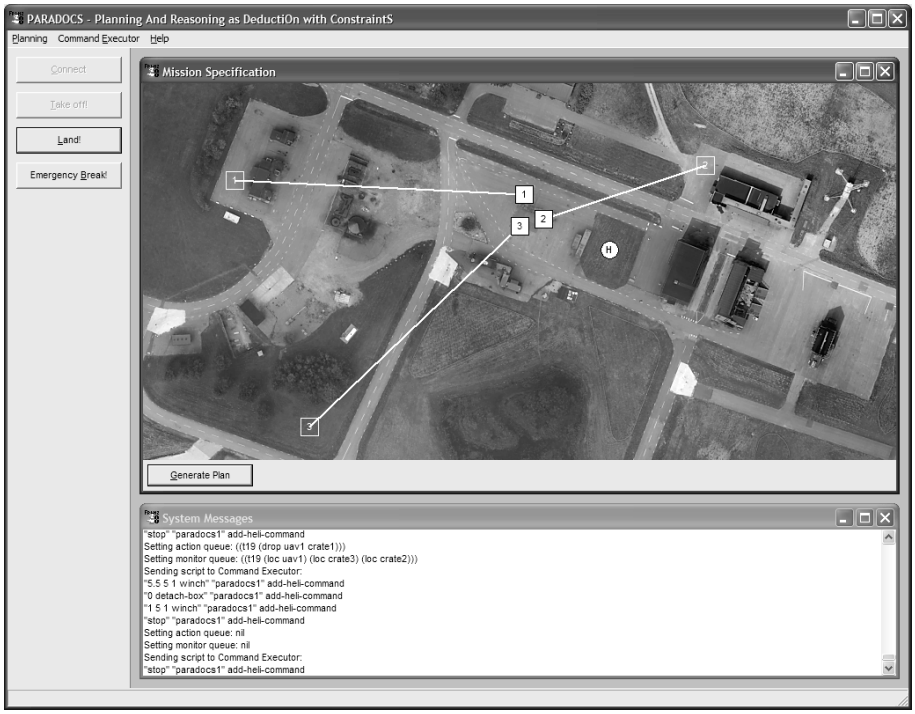

Figure 6.1: A logistics mission in the graphical user interface.

them. These enhancements would make it possible to perform the scenario using the helicopter hardware.

6.1 Graphical User Interface

The graphical user interface is implemented in Lisp using the Franz Allegro CL Common Graphics multi platform GUI library [15]. It provides a window based interface with an aerial overview of the flying area overlayed by icons representing objects of interest. Setting up a logistics mission simply consists of dragging crate icons to indicate their intended goal locations. Consider e.g. the mission set up in Figure 6.1. The task is to deliver three crates from a common store to three different locations. Clicking the “Generate Plan”

button will call on SICStus Prolog 3.12.5 [30] to evaluate the logic program (found in the appendix) on the following goal generated by the interface:

```
:- holds(T,loc(crate3),pos(113,-96),P),
   holds(T,loc(crate1),pos(82,6),P),
   holds(T,loc(crate2),pos(277,12),P),
   timepoint_variables(L), labeling([],L).
```

Iterative deepening on timepoint variable domains would generate a plan with the smallest consistent temporal network. Such an exhaustive search would be completed on an Intel Pentium M 1.8 GHz within thirteen minutes. But the goal-directed search exhibited by PARADOCS is able to find plans in the logistics scenario without a depth bound. The plans are not guaranteed to be optimal, but they avoid the complete exploration of the search space on each depth bound smaller than the one that produces the first plan. With this approach PARADOCS produces a plan consisting of 12 actions and 15 persistence intervals in a graph containing a total of 22 temporal intervals in less than a tenth of a second.

The need for a graphical interface was evident even during the development and experimentation with deductive planning and temporal constraints. The constraint networks are represented as predicate lists by the constraint solver. These lists quickly grow unreadably large, prompting some form of visualization tool for grasping the temporal relations between action intervals and fluent persistence intervals. The first of two such visualizations in the GUI is an interval algebra graph that displays the partially specified temporal relations between intervals. Figure 6.2 shows the solution graph for the above logistics mission. The interval algebra representation's support for partial knowledge in the form of disjunctions makes it impossible to visualize the order in which the actions of the plan will be executed. In fact, at this point the order has not yet been decided. Still, it is possible to partially order the graph so that intervals higher up in the display are known to occur before intervals below them.

Even though transitive and reflexive relations are hidden the graph is still difficult to decipher for all but the tiniest of plans. Instead, the operator clicks a "Next Linearization" button to apply a backtracking labeling algorithm that returns all possible linearizations of the partial order plan, one at a time. At this point visualization is much easier as the order of actions has been decided. The plan chart in Figure 6.3 shows the first linearization of the above graph.

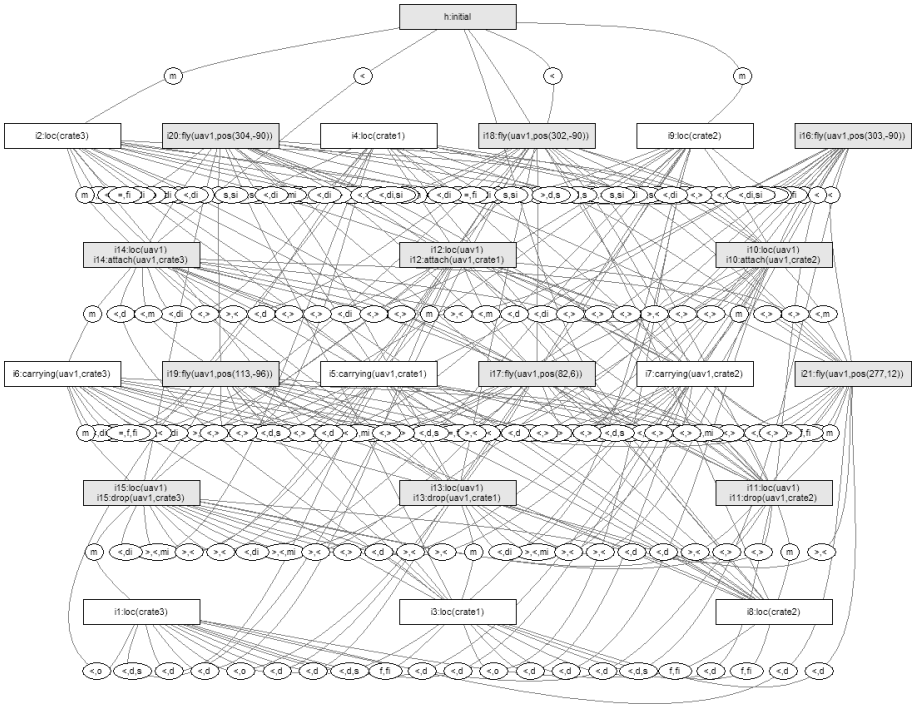

Figure 6.2: An interval algebra graph visualization of a partially ordered plan.

6.2 Execution and Monitoring

The GUI provides an interface for an operator of the UAV to the PARADOCS planning service, which is part of the software system that controls the robot platform [6]. The software system implements a deliberative/reactive architecture where components use the CORBA framework to call each other's methods. This enables a highly distributed system in which components can use different implementation languages and reside on different computers as long as they are connected in a network.

Planned actions are stored in an action queue. They are popped from the queue during execution and communicated to a Command Executor service that assumes responsibility for calling the appropriate low-level control functionalities to execute each action. The planned actions are guaranteed to be consistent with the persistence assumptions that were made during the plan-

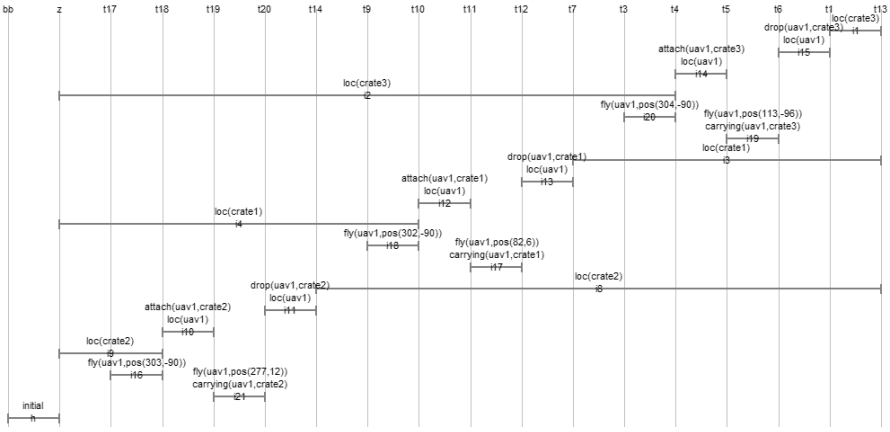

Figure 6.3: A plan chart showing one of many possible linearizations of the plan.

ning process, but unexpected things can happen during execution. The plan's persistence constraints, in the form of non-occlusion formulas that must hold during specific time intervals, are therefore placed in a monitor queue in parallell with the action queue. Monitor constraints that are active during an interval are evaluated by regularly querying the simulator for the values of the fluents and making sure that the values are indeed persistent and do not change.

Assume however, for the sake of example, that the electro-magnet device looses hold of *crate1* while the UAV is flying towards its destination in the middle of executing the plan in Figure 6.3. The planning process had already recorded the assumption that the fluent *carrying(uav1,crate1)* is persistent during the flight. This is necessary to satisfy the conditions for the action of dropping *crate1* at its destination. But when the hold of the crate is lost, the truth value of the *carrying* fluent changes to false, which violates said monitoring formula. At this point the plan is no longer guaranteed to achieve the goal and execution is halted.

Since PARADOCS can be applied equally well to problems in-between planning and prediction we can use the part of the plan that has not yet been executed as input for a plan repair process. The following goal is automatically generated and passes the actions that are left to execute as input through the action occurrence argument and ensures that they satisfy the previously

selected linearization using constraints on the timepoint variables:

```
:- P = [occ(T1,T2,fly(uav1,pos(82,6))),
        occ(T2,T3,drop(uav1,crate1)),
        occ(T4,T5,fly(uav1,pos(175,-35))),
        occ(T5,T6,attach(uav1,crate3)),
        occ(T6,T7,fly(uav1,pos(113,-96))),
        occ(T7,T8,drop(uav1,crate3))|Pp],
   T1 #< T2, T2 #< T3, T3 #< T4, T4 #< T5,
   T5 #< T6, T6 #< T7, T7 #< T8,
   holds(Tn,loc(crate3),pos(113,-96),P),
   holds(Tn,loc(crate1),pos(82,6),P),
   timepoint_variables(L), labeling([],L).
```

The result is, in this case, the insertion of a `fly(uav1,pos(153,2))` and an `attach(uav1,crate1)` action in front of the rest of the plan. These actions constitute a recovery by going back to pick up the crate that was accidentally dropped before continuing with the rest of the plan.

Chapter 7

Related Work

Much of the inspiration to our work comes from other research groups with similar approaches. We list here only the most important influences and start with Shanahan's abductive Event Calculus planner [33]. It is based on the Event Calculus, a logical formalism for reasoning about action and change that, like TAL, uses an explicit time line and an *Occurs* predicate to link action occurrences to the time line. This makes it suitable for the generation of partially ordered plans and, as with PARADOCS, practical planning is accomplished using logic programs. However, plan synthesis is based on abduction of *Occurs* instead of deduction with reified action occurrences. The use of abduction results in theoretical elegance, but performing abduction in a logic program requires an abductive meta-interpreter, which makes the programs more complex.

The timepoint relations of the abductive Event Calculus planner can only detect inconsistencies caused by conflicts between action preconditions and action effects if the order of timepoints involved in the conflict is known. Consistency is guaranteed by adding new timepoint relations in what could be described as the logical equivalent of the promote/demote strategy from partial order planning algorithms (page 12, [33]). But resolving potential conflicts through promotion or demotion represents an early commitment to a specific ordering, and both orderings must be evaluated in cases where the planning process backtracks. Our constraint logic programs take full advantage of the representational power of disjunctive interval algebra relations. Disjunctive non-overlap constraints allow the planner to postpone decisions on the ordering of actions, even in the case of a possible conflict, thereby reducing the search space. We have in fact experimented with weaker constraint solvers similar to

simple temporal networks, but found the added complexity of the implementation that result from the introduction of promotion and demotion to detract from the clarity of presentation. Although the use of disjunctive constraints is associated with an increase in the computational complexity of the constraint propagation algorithms, the complexity is still overshadowed by the complexity of planning in general.

In a feature comparison, the Event Calculus planner extends our basic planning capabilities with hierarchical planning and a form of knowledge producing sensing actions that are also based on abduction [34].

Another *deductive* planning framework is GOLOG [21], which extends the Situation Calculus with composite actions, including the sequences, conditionals, non-deterministic choices, and loops that we introduced in Chapter 5. The paradigm is slightly different in that it views the resulting formalism as a high-level agent *programming language* where the programmer supplies a possibly incomplete program specification, and the robot's task is to execute the specification while resolving non-determinism that may result from its incompleteness.

The Situation Calculus originated the idea of reified action occurrences passed around using an extra predicate argument. But its situation terms contain linearly ordered action sequences without explicit temporal information, which prevents the generation of partially ordered plans. Note however, that such shortcomings can be overcome through various extensions, as is done e.g. in ConGolog [11]. Both the Situation Calculus and the GOLOG framework have been extended in a number of other interesting ways, e.g. for planning with incomplete information about the initial state [9].

While GOLOG uses composite actions to provide a form of search control through domain dependant knowledge, complex actions are not part of the planning process in the way that simple actions are. This is no great surprise since, as we noted in Section 5.4, the resulting system would in fact perform program synthesis. However, Levesque's KPLANNER [20] explores an interesting middleground between the direct synthesis of programs with loops and GOLOG's execution of non-deterministic loops supplied by a programmer. It is applicable to problems where it is possible to identify a single fluent, whose value is unknown or unbound, that is responsible for making the problem unsolvable without the generation of plans containing loops. The method consists of setting a relatively small upper bound on the value of this fluent, generating a plan with help from a clever Prolog function that "winds up" action sequences into loops, and finally testing this plan for a larger bound (since testing will be cheaper than generating). Using this method one can generate plans with loops that are correct up to the testing bound, and, for some classes of problems,

provably correct for any value of the unbounded fluent.

Another approach to the use of composite actions in planning is presented by McIlraith and Fadel [27] who were motivated by the needs of planning for the use of semantic web services, but also by the increased plan generation efficiency resulting from the possibility of shorter solution plans. They provide a compilation, expressed in terms of the Situation Calculus, of action sequences, conditionals, and loops, into a new set of actions that are not composite. However, due to the requirements enforced on the composite actions to ensure that the compilation is possible, only bounded loops with a predetermined maximum number of iterations are considered. Our PARADOCS framework does not provide a means for the direct use of composite actions in planning, but our loop semantics, expressed in terms of a fixpoint formula, is not associated with any preset bounds.

The Fluent Calculus serves as the formal basis for FLUX [37], another logic programming methodology that supports deductive planning with linear plans. It was mentioned earlier as the source of the Prolog list membership function implementation used for adding reified action occurrence terms to a list. In FLUX the same predicate is used for a different purpose as the basic mechanism of representing states as sets of fluents. Constraint handling rules are also used in FLUX, but again for different purposes than ours. While we utilize CHR for detecting and resolving potential conflicts between action effects and fluent persistency, in FLUX they enable the planning and reasoning with incomplete information and also reasoning about knowledge and knowledge updates as caused by sensing actions. Although the form of the expressions over which such reasoning is allowed is limited, the limitations ensure that the allowed reasoning is computationally efficient.

Our final example of logic programming based planning is answer set planning, as described e.g. by Lifschitz [23]. Answer set programming is purely declarative, and thus avoids some of the problems inherent in Prolog, such as the possibility of the evaluation becoming stuck in an infinite loop. This property makes it easier to add temporal constraints and state constraints on the answer generation. Lifschitz proposes that its use of both classical negation and negation as failure makes it suitable for specifying the effects of actions and the non-monotonic behaviour of the persistence of fluents that they affect.

In addition, Son, Baral, and McIlraith [35] show how answer set programming can be extended with sequences, conditionals, non-deterministic choice, and loops, to construct an alternative GOLOG interpreter. This can be used, in the same way as any Prolog GOLOG interpreter, to write non-deterministic programs that, in effect, express domain-specific control knowledge restricting the search for solution plans.

However, the evaluation mechanism of answer set programs includes instantiating the problem into a finite number of propositions, and is thus only applicable to finite theories with a fixed upper time bound. While such a "propositional" approach, also used in the planning as satisfiability paradigm, may be simple and robust, it can suffer from an exponential increase in the size of the representation of the problem. It also lacks the goal-directedness of Prolog's evaluation mechanism, based on theorem proving, that seems to us to be a desireable property that will be absolutely necessary as the complexity of problems a robot attempts to solve increases, and as the number of different means it has available to apply to the problems grows.

An even stronger programming language focus is displayed by agent programming languages such as 3APL [14]. The concept of a goal is considered to be procedural rather than declarative, and actions are described as state updates that modify an agent's beliefs, rather than being defined by sets of axioms. While 3APL provides a formal semantics, it is an operational semantics distinct from the programming language itself. Although the procedural view of goals might seem restrictive, from the point of view of planning, it is surprisingly close to e.g. GOLOG's programs. In fact, it is possible to embed ConGolog *into* 3APL [13].

Another system with a focus on planning, and with many features in common with PARADOCS, is Allen's temporal planner [2]. It expresses temporal information using the interval temporal logic that axiomatizes the interval algebra and extends it with atomic time periods that behave like timepoints and represents actions by reified events. Allen also stresses the importance of viewing different types of reasoning as inference in a common representation, although, the actual plan synthesis is cast as an algorithm that closely approximates a proof procedure.

A different view is endorsed in planners that utilize special purpose planning algorithms. TALplanner [7] is one such planner that is especially close to ours in that it makes use of the same Temporal Action Logic. However, the purpose of TAL in TALplanner is as a formal semantics of actions, goals, and plans. Employing Temporal Action Logic as a formal semantics for the actual *planning process*, and using it directly for plan synthesis, prompted the extensions and work described in this thesis.

Karlsson proceeds with a more theoretical emphasis when he presents his Narrative Logic, based on TAL but with its own extensions for reified action occurrences [16]. Karlsson formulates the planning task in Narrative Logic but, although he provides a translation from Narrative Logic into classical logic, the result is a second-order theory that is not directly amenable to automated theorem proving. In contrast, our translation of TAL with reified action occurrences

into logic programs makes the logic directly applicable in practice.

Both Karlsson's work and ours extend TAL, which itself originates in Sandewall's *Features and Fluents* framework [32]. In fact, Sandewall himself provides a formulation of composite actions, including sequences, conditionals, and loops, whose macro-language syntax is almost identical to ours. However, the translation of loops differs significantly. Sandewall uses a first-order definition involving special *signal features*, which are introduced only to keep track of the looping behaviour. This results in a specification that is rather complex, whereas our formulation in terms of a fixpoint formula is significantly simpler while succinctly capturing the looping property.

Furthermore, we extend Sandewall's set of composite actions with nondeterministic choice, which we found necessary for composite actions to be of practical use, and we make such use possible through the provision of our compilation into logic programs.

Chapter 8

Discussion

Although Temporal Action Logic has long served as a formal basis for some of our work in cognitive robotics, we set out to use TAL for deductive planning. This effort necessitated some theoretical additions to the logic but has also resulted in a compilation process that extends the translation from the high-level language $\mathcal{L}$(ND) to constraint logic programs that are efficiently executed for practical planning. The persistence constraints that are part of the generated plans can be used for execution monitoring purposes and the integration of plan synthesis and reasoning about plans is put to use in the failure recovery process. The entire set up is integrated with our UAV system and can be used through a graphical user interface with drag'n'drop mission planning to execute logistics missions in our helicopter simulator that adds a winch and crates to our fully operational UAV research platform.

The explicit time representation of TAL exposes qualitative and quantitative temporal primitives that are particularly amenable to reasoning using temporal constraint formalisms. The method can be made to work with a variety of constraint formalisms for both qualitative and quantitative constraints, such as simple timepoint constraints, interval algebra, or general temporal constraint networks. But the combination of constraint handling rules and disjunctive temporal constraints is particularly suited to the TAL formalism. This, together with the occlusion concept, enables a novel solution to potential action/persistence conflict threats where promotion/demotion is postponed, thereby removing search space choice points and resulting in flexible solution plans.

8.1 Future Research

While compiling narratives into logic programs that can be used for deductive planning works very well, the technique is ultimately limited. As stated in the introduction Section 1.1, our two main motivations for planning deductively are uniformity and expressiveness. But the expressivity of logic programs are inherently limited, and each attempt to circumvent specific limitations using additional special purpose techniques will result in decreasing uniformity. For these reasons we believe a move towards a more direct use of theorem proving in the first-order base logic of TAL to be important.

A move from logic programming towards more general theorem proving will surely lead to inefficiency and scalability problems. The resulting system might only be able to reason with relatively small problem instances and complexity might prevent it from scaling up to problem instances of a larger size that could have been solved by a less expressive system. Though, the class of problems to which the system is applicable would be larger. And when that class expands to encompass the generation of plans with loops or recursion, as discussed in Section 5.4, the rules of the game suddenly change. Consider e.g. a logistics scenario where the number of crates to be delivered is unspecified. Any solution plan will have to contain some form of looping or recursion behaviour that iterates over the crates until the goal has been satisfied. Plans of this highly expressive sort can probably be small and compact. Moreover, the plan is of constant size, *regardless* of the size of the logistics problem instance that it is later applied to. Finally, the solution simultaneously solves all problem instantiations with any number of crates. In contrast, a less expressive planner would not be applicable to a logistics problem where it is unknown what crates exists. It would exhibit an often exponential, but at least linear, increase in planning time and a linear increase in solution length as the size of the problem instance grows. Finally, it would require more planning and the generation of a new plan every time a new problem instance with a different number of crates needs to be solved.

The problem of generating plans with loops or recursion is a very difficult problem to which deductive planning with theorem proving techniques has been applied with some success [26, 3, 4]. In addition to the "plans as programs" paradigm, deductive planning with a highly expressive representation is potentially applicable to a large number of other interesting problems. Two of the most important are planning in the context of incomplete information and reasoning about knowledge and knowledge-producing sensing actions. Both are possible continuations of the work in this thesis.

8.2 Conclusions

Nilsson [29] argues for a distinction between general intelligent systems and specialized systems that exhibit greater than human performance in a relatively narrow area of expertise. A logical framework for reasoning and planning with actions is, by design, aimed towards generality rather than performance when compared with a special purpose planner. This thesis presents a methodology for practical deductive planning in TAL that is a step towards a uniform way of tackling increasing complexity. Moving towards increasingly intelligent autonomous systems means requring less help from humans who predict and decide what tools need to be applied in solving a given task. At the same time, the increasing diversity and challenge of the tasks themselves require the application of flexible and powerful reasoning methods. By reformulating special-purpose algorithms as proof search in a shared representational formalism, one removes the need to decide in advance what algorithms are needed to solve future tasks. Through the use of highly expressive logical formalisms and automated theorem proving technology one opens the possibility of attacking a great many different and complex reasoning problems in a uniform way. Although this unifying approach might not have been the way of the (recent) past, the growing challenges of artificial intelligence applications is increasing its appeal as the way of the future.

Appendix A

The Logistics Scenario Prolog Code

This appendix contains the UAV logistics scenario constraint logic program encoding that has been integrated with the autonomous helicopter system and was used for the planning examples in Chapter 6. Readers interested in running the program can also access this code in electronic form at this thesis' website: `http://www.martinmagnusson.com/paradocs/thesis/`.

Three additional comments will assist understanding of the code. Firstly, the persistence clauses have a potential problem with infinite looping given the depth-first search strategy of Prolog. A fluent is true at a timepoint if it is true at an earlier timepoint and is not occluded during the interval in between. It is true at the earlier timepoint if it is true at an even earlier timepoint, and so on. The looping is correct but unwanted behaviour. But the observation that if a fluent is persistent over two intervals that meet, then it must be persistent over the union of the intervals, provides the key to a simple solution. Without loss of generality we prohibit two consecutive persistence intervals using an additional constraint that appears in the code below with the comment "Consecutive persistence constraint".

Secondly, although there is only one UAV in our scenario there are few restrictions on the occurrence times of actions, which can lead to modelling problems related to concurrency. The UAV magnetic hook can be seen as a limited resource that can not attach several crates at once. However, modelling this using a fluent precondition *free* that is set to false when executing *attach* does not prevent two concurrent *attach* at the *exact* same time. Actions can only have effects in the future, so one *attach* can not prevent another *attach*

from executing simultaneously by setting *free* to *false* at the next timepoint. Gustafsson [12] suggests a way of solving this problem using TAL narratives where actions do not have direct effects on fluents, but rather activate special *influence* fluents. Dependency constraints are then used to express influence laws that govern the effects of influences on the environment, and thereby the indirect effects of actions. Concurrent planning in TAL, using an explicit time line and powerful temporal constraints solvers, holds great promise but clearly needs more investigation. We avoid the issue for now and instead add two concurrency constraints even though they are not part of the translation specified in Chapter 4. Specifically, the constraints with the comment "Concurrency constraints" prohibit the UAV from carry several crates and from flying to several places concurrently.

Thirdly, and finally, at the same time that non-overlap constraints are added to the finite domain constraint store, they are stored in the CHR store using a constraint named `ia`. The final group of clauses, which appear below the comment "Backup store", are then used to create a copy of the CHR constraint store before the labeling of timepoint variables that determines if the interval algebra network is consistent. This prevents the labeling process from instantiating and unifying timepoint variables in the network copy while searching for a consistent linearization of the original, thereby preserving all of the partialness of the ordering of action occurrences. The final interval algebra graph is then easily read from the `ia` constraint in the constraint store copy, and it is guaranteed to be consistent since the original went through a labeling.

```
:- use_module(library(chr)).
:- use_module(library(clpfd)).
:- use_module(library(charsio)).

% CHR definitions:
handler tal.
constraints not_occlude/3, circ_occurs/3, tp/1, ia/3.

% Occurs definition:
occurs(T1,T2,A,P) :-
  member(occ(T1,T2,A),P), circ_occurs(T1,T2,A).
member(A,[A|Pp]).
member(A,P) :-
  nonvar(P), P = [A1|P1], A \== A1, member(A,P1).

% Interval relations:
not_overlap(Xs,Xe,Ys,Ye) :-
```

```
  Xe #=< Ys #\/ Xs #>= Ye,
  ia(Xs-Xe,[<,m,mi,>],Ys-Ye).

% Type clauses:
timepoint(T) :- domain([T],0,100), tp(T).
uav(uav1).
crate(crate1).
crate(crate2).
crate(crate3).

% Persistence formulas:
holds(T2,loc(X),V,P) :-
  timepoint(T1), timepoint(T2), T1 #< T2,
  not_occlude(T1,T2,loc(X)),
  holds(T1,loc(X),V,P).
holds(T2,carrying(U,C),V,P) :-
  timepoint(T1), timepoint(T2), T1 #< T2,
  not_occlude(T1,T2,carrying(U,C)),
  holds(T1,carrying(U,C),V,P).

% Consecutive persistence constraint:
not_occlude(T1,T,F), not_occlude(T,T2,F) ==> fail.

% Initial state (example):
holds(0,loc(uav1),pos(237,-23),P).
holds(0,loc(crate1),pos(202,0),P).
holds(0,loc(crate2),pos(210,-10),P).
holds(0,loc(crate3),pos(200,-13),P).

% Action specifications:
holds(T2,loc(U),L,P) :-
  uav(U), timepoint(T1), timepoint(T2), T1 #< T2,
  occurs(T1,T2,fly(U,L),P).
not_occlude(T1,T2,loc(U)),
  circ_occurs(T3,T4,fly(U,L)) ==>
    not_overlap(T1,T2,T3,T4).

holds(T2,carrying(U,C),true,P) :-
  uav(U), crate(C), timepoint(T1), timepoint(T2), T1 #< T2,
  occurs(T1,T2,attach(U,C),P),
```

```
  holds(T1,loc(C),L,P),
  holds(T1,loc(U),L,P),
  not_occlude(T1,T2,loc(U)).
not_occlude(T1,T2,carrying(U,C)),
  circ_occurs(T3,T4,attach(U,C)) ==>
    not_overlap(T1,T2,T3,T4).
not_occlude(T1,T2,loc(C)),
  circ_occurs(T3,T4,attach(U,C)) ==>
    not_overlap(T1,T2,T3,T4).

holds(T2,loc(C),L,P) :-
  uav(U), crate(C), timepoint(T1), timepoint(T2), T1 #< T2,
  occurs(T1,T2,drop(U,C),P),
  holds(T1,carrying(U,C),true,P),
  holds(T1,loc(U),L,P),
  not_occlude(T1,T2,loc(U)).
not_occlude(T1,T2,carrying(U,C)),
  circ_occurs(T3,T4,drop(U,C)) ==>
    not_overlap(T1,T2,T3,T4).
not_occlude(T1,T2,loc(C)),
  circ_occurs(T3,T4,drop(U,C)) ==>
    not_overlap(T1,T2,T3,T4).

% Concurrency constraints:
not_occlude(T1,T2,carrying(U,C1)),
  not_occlude(T3,T4,carrying(U,C2)) ==>
    not_overlap(T1,T2,T3,T4).
circ_occurs(T1,T2,fly(U,L1)),
  circ_occurs(T3,T4,fly(U,L2)) ==>
    not_overlap(T1,T2,T3,T4).

% Remove redundant constraints:
not_occlude(T1,T2,F) \ not_occlude(T1,T2,F) <=> true.
circ_occurs(T1,T2,A) \ circ_occurs(T1,T2,A) <=> true.
tp(X) \ tp(X) <=> true.
tp(X) <=> ground(X) | true.

% Find variables to label:
timepoint_variables(L) :-
  findall_constraints(tp(_),L1),
```

```
  extract_variable(L1,L).
extract_variable([],[]).
extract_variable([tp(T)#_|L1],[T|L2]) :-
  extract_variable(L1,L2).

% Backup store:
copy_constraints(C) :-
  findall_constraints(_,L1),
  extract_constraint(L1,L),
  format_to_chars('~w',[L],C).
extract_constraint([],[]).
extract_constraint([C#_|L1],[C|L2]) :-
  extract_constraint(L1,L2).
```

Appendix B

Proofs

This appendix collects proofs of some theorems used in the thesis.

B.1 Interval Persistence Formula

The interval persistence formula relates regular, single timepoint, occlusion with the new interval occlusion.

Theorem

$$\begin{aligned}&\forall t\,[\neg Occlude(t+1,f) \rightarrow \forall v\,[Holds(t,f,v) \leftrightarrow Holds(t+1,f,v)]] \wedge \\ &\forall t_1,t_2,f\,[Occlude(t_1,t_2,f) \leftrightarrow \exists t\,[t_1 < t \leq t_2 \wedge Occlude(t,f)]] \rightarrow \\ &\forall t_1,t_2,f\,[\neg Occlude(t_1,t_2,f) \rightarrow \\ &\qquad\qquad \forall t\,[t_1 < t \leq t_2 \rightarrow \forall v\,[Holds(t-1,f,v) \leftrightarrow Holds(t,f,v)]]]\end{aligned}$$

Proof
Given $\neg Occlude(t_1,t_2,f)$ we know, by the definition of interval occlusion above, that there does not exist a timepoint t during the interval $(t_1,t_2]$ at which $Occlude(t,f)$ holds, i.e., $\neg Occlude(t,f)$ must hold for all timepoints in the interval. The persistence formula then forces the fluent to retain its value from the previous timepoint, which is specified in the above theorem by stating that the *Holds* predicate retains its truth value for all possible fluent values v. We formalize this reasoning in a deductive proof below.

1	$\forall t\,[\neg Occlude(t+1,f) \rightarrow \forall v\,[Holds(t,f,v) \leftrightarrow Holds(t+1,f,v)]]$	P
2	$\forall t_1,t_2,f\,[Occlude(t_1,t_2,f) \leftrightarrow \exists t\,[t_1 < t \leq t_2 \wedge Occlude(t,f)]]$	P
3	$\neg Occlude(t_1,t_2,f) \rightarrow \neg\exists t\,[t_1 < t \leq t_2 \wedge Occlude(t,f)]$	2

4	$\neg Occlude(t_1, t_2, f)$	H
5	$\neg\exists t\,[t_1 < t \leq t_2 \wedge Occlude(t, f)]$	$3, 4$
6	$\forall t\,[t_1 < t \leq t_2 \rightarrow \neg Occlude(t, f)]$	5
7	$t_1 < t \leq t_2 \rightarrow \neg Occlude(t, f)$	6
8	$t_1 < t \leq t_2$	H
9	$\neg Occlude(t, f)$	$7, 8$
10	$\neg Occlude((t-1)+1, f)$	9
11	$\neg Occlude((t-1)+1, f) \rightarrow$ $\forall v\,[Holds(t-1, f, v) \leftrightarrow Holds((t-1)+1, f, v)]$	1
12	$\forall v\,[Holds(t-1, f, v) \leftrightarrow Holds((t-1)+1, f, v)]$	$10, 11$
13	$\forall v\,[Holds(t-1, f, v) \leftrightarrow Holds(t, f, v)]$	12
14	$\forall t\,[t_1 < t \leq t_2 \rightarrow \forall v\,[Holds(t-1, f, v) \leftrightarrow Holds(t, f, v)]]$	$8 - 13$
15	$\forall t_1, t_2, f\,[\neg Occlude(t_1, t_2, f) \rightarrow$ $\forall t\,[t_1 < t \leq t_2 \rightarrow Holds(t-1, f) \leftrightarrow Holds(t, f)]]$	$4 - 14$

B.2 Interval End-point Equivalence

The following proof validates the useful property of interval occlusion that a fluent that is known not to be occluded during an interval will have the same value at both end-points of that interval, regardless of the number of timepoints inbetween.

Theorem

$$\forall t_1, t_2, f\,[\neg Occlude(t_1, t_2, f) \rightarrow \forall t\,[t_1 < t \leq t_2 \rightarrow \forall v\,[Holds(t-1, f, v) \leftrightarrow Holds(t, f, v)]]] \wedge$$
$$\forall t_1, t_2, f\,[Occlude(t_1, t_2, f) \leftrightarrow \exists t\,[t_1 < t \leq t_2 \wedge Occlude(t, f)]] \rightarrow$$
$$\forall t_1, t_2, f\,[\neg Occlude(t_1, t_2, f) \rightarrow \forall v\,[Holds(t_1, f, v) \leftrightarrow Holds(t_2, f, v)]]$$

Proof

The proof is by induction over the length of occlusion intervals. The base case consists of intervals of length 1. Such intervals are already covered by the sub-formula $\forall v\,[Holds(t-1, f, v) \leftrightarrow Holds(t, f, v)]$ in the interval persistence formula above, which we make use of in the deduction below.

16	$\forall t_1, t_2, f\,[\neg Occlude(t_1, t_2, f) \rightarrow$ $\forall t\,[t_1 < t \leq t_2 \rightarrow \forall v\,[Holds(t-1, f, v) \leftrightarrow Holds(t, f, v)]]]$	P

17 ┌ $\neg Occlude(t_1, t_2, f)$ *H*
18 | $\forall t\,[t_1 < t \leq t_2 \rightarrow \forall v\,[Holds(t-1, f, v) \leftrightarrow Holds(t, f, v)]]$ 16, 17
19 | $t_2 = t_1 + 1$ the interval has length 1
20 | $\forall t\,[t_1 < t \leq t_1 + 1 \rightarrow \forall v\,[Holds(t-1, f, v) \leftrightarrow Holds(t, f, v)]]$ 18, 19
21 | $t_1 < t_1 + 1 \leq t_1 + 1 \rightarrow$
| $\quad \forall v\,[Holds((t_1 + 1) - 1, f, v) \leftrightarrow Holds(t_1 + 1, f, v)]$ 20
22 | $\forall v\,[Holds(t_1, f, v) \leftrightarrow Holds(t_1 + 1, f, v)]$ *F*, 21
23 | $\forall v\,[Holds(t_1, f, v) \leftrightarrow Holds(t_2, f, v)]$ 19, 22
24 $\forall t_1, t_2, f\,[\neg Occlude(t_1, t_2, f) \rightarrow$
$\quad \forall v\,[Holds(t_1, f, v) \leftrightarrow Holds(t_2, f, v)]]$ 17 – 23

In the inductive case we assume that the theorem holds for intervals of length n and show that it holds for intervals of length $n + 1$. First note that, by the definition of interval occlusion, if the fluent is interval occluded over an interval $(t, t + n + 1]$, of length $n + 1$, then it must also be interval occluded over the sub-intervals $(t, t + n]$ and $(t + n, t + n + 1]$, of length n and 1 respectively. Since we assumed that the theorem holds for intervals of length n, the fluent will have the same value at t and $t + n$. As with the interval of length 1 in the base case, the interval persistence formula forces the fluent to retain its value between $t + n$ and $t + n + 1$. Taken together these two conditions establish the conclusion that the fluent will necessarily have the same value at both end-points of intervals of length $n+1$. This is formalized, somewhat cumbersomely, in the deduction below.

25 $\forall t_1, t_2, f\,[\neg Occlude(t_1, t_2, f) \rightarrow$
$\quad \forall t\,[t_1 < t \leq t_2 \rightarrow \forall v\,[Holds(t-1, f, v) \leftrightarrow Holds(t, f, v)]]]$ *P*
26 $\forall t_1, t_2, f\,[Occlude(t_1, t_2, f) \leftrightarrow \exists t\,[t_1 < t \leq t_2 \wedge Occlude(t, f)]]$ *P*
27 ┌ $\neg Occlude(t_1, t_2, f)$ *H*
28 | $t_2 = t_1 + n + 1$ the interval has length $n + 1$
29 | $\neg Occlude(t_1, t_2, f) \rightarrow \neg\exists t\,[t_1 < t \leq t_2 \wedge Occlude(t, f)]$ 26
30 | $\neg\exists t\,[t_1 < t \leq t_2 \wedge Occlude(t, f)]$ 27, 29
31 | $\forall t\,[t_1 < t \leq t_2 \rightarrow \neg Occlude(t, f)]$ 30
32 | ┌ $t_1 < t \leq t_1 + n$ *H*
33 | | $t_1 < t \leq t_2 - 1$ 28, 32
34 | | $t_1 < t \leq t_2$ 33, *F*
35 | | $\neg Occlude(t, f)$ 31, 34
36 | $t_1 < t \leq t_1 + n \rightarrow \neg Occlude(t, f)$ 32 – 35

37	$\forall t\,[t_1 < t \leq t_1 + n \rightarrow \neg Occlude(t, f)]$	36
38	$\neg\exists t\,[t_1 < t \leq t_1 + n \wedge Occlude(t, f)]$	37
39	$\neg Occlude(t_1, t_1 + n, f)$	26, 38
40	$\forall v\,[Holds(t_1, f, v) \leftrightarrow Holds(t_1 + n, f, v)]$	39, inductive assumption
41	$\quad t_1 + n < t \leq t_1 + n + 1$	H
42	$\quad t_1 < t \leq t_1 + n + 1$	41, F
43	$\quad t_1 < t \leq t_2$	28, 42
44	$\quad \neg Occlude(t, f)$	31, 43
45	$t_1 + n < t \leq t_1 + n + 1 \rightarrow \neg Occlude(t, f)$	$41 - 44$
46	$\forall t\,[t_1 + n < t \leq t_1 + n + 1 \rightarrow \neg Occlude(t, f)]$	45
47	$\neg\exists t\,[t_1 + n < t \leq t_1 + n + 1 \wedge Occlude(t, f)]$	46
48	$\neg Occlude(t_1 + n, t_1 + n + 1, f)$	26, 47
49	$\forall t\,[t_1 + n < t \leq t_1 + n + 1 \rightarrow \forall v\,[Holds(t - 1, f, v) \leftrightarrow Holds(t, f, v)]]$	25, 48
50	$t_1 + n < t_1 + n + 1 \leq t_1 + n + 1$	F
51	$\forall v\,[Holds(t_1 + n, f, v) \leftrightarrow Holds(t_1 + n + 1, f, v)]$	49, 50
52	$\forall v\,[Holds(t_1, f, v) \leftrightarrow Holds(t_2, f, v)]$	28, 40, 51
53	$\forall t_1, t_2, f\,[\neg Occlude(t_1, t_2, f) \rightarrow \forall v\,[Holds(t_1, f, v) \leftrightarrow Holds(t_2, f, v)]]$	$27 - 52$

By the principle of mathematical induction, the theorem holds for intervals of any length.

B.3 Point-interval Rewrite

This particular equivalence-preserving rewrite is used in the compilation of a TAL theory to replace timepoint occlusion formulas with equivalent formulas expressed using interval occlusion.

Theorem

$\forall t\,[P(t) \rightarrow Q(t)] \leftrightarrow$
$\forall t_1, t_2\,[\exists t\,[t_1 < t \leq t_2 \wedge P(t)] \rightarrow \exists t\,[t_1 < t \leq t_2 \wedge Q(t)]]$

Proof

We start with the $\Rightarrow$ direction and assume that the left hand side holds. Assume further that there is some t in the interval $(t_1, t_2]$ for which P holds, i.e.,

$\exists t\,[t_1 < t \leq t_2 \wedge P(t)]$. Apply the assumed implication $P(t) \rightarrow Q(t)$ to this t to show the existence of a timepoint in the interval $(t_1, t_2]$ for which Q holds, i.e., $\exists t\,[t_1 < t \leq t_2 \wedge Q(t)]$. For the $\Leftarrow$ direction, assume that the right hand side holds. Assume further that $P(t')$ holds, for an arbitrary timepoint t'. Instantiate the universally quantified t_1 and t_2, in the right hand side of the theorem, to $t' - 1$ and t' respectively. Since t' is in the interval $(t' - 1, t']$, and we assumed $P(t')$, we have $\exists t\,[t' - 1 < t \leq t' \wedge P(t)]$ and conclude $\exists t\,[t' - 1 < t \leq t' \wedge Q(t)]$. The narrow interval permits only one value for the existentially quantified timepoint t, namely t', and we thus have $Q(t')$. Since t' was arbitrary, we conclude that $\forall t\,[P(t) \rightarrow Q(t)]$. The argument is restated formally below.

54	$\forall t\,[P(t) \rightarrow Q(t)]$	H
55	$\exists t\,[t_1 < t \leq t_2 \wedge P(t)]$	H
56	$t_1 < t \leq t_2 \wedge P(t)$	H
57	$P(t) \rightarrow Q(t)$	54
58	$Q(t)$	56, 57
59	$t_1 < t \leq t_2 \wedge Q(t)$	56, 58
60	$\exists t\,[t_1 < t \leq t_2 \wedge Q(t)]$	59
61	$\exists t\,[t_1 < t \leq t_2 \wedge Q(t)]$	$55, 56 - 60$
62	$\forall t_1, t_2\,[\exists t\,[t_1 < t \leq t_2 \wedge P(t)] \rightarrow \exists t\,[t_1 < t \leq t_2 \wedge Q(t)]]$	$55 - 61$
63	$\forall t_1, t_2\,[\exists t\,[t_1 < t \leq t_2 \wedge P(t)] \rightarrow \exists t\,[t_1 < t \leq t_2 \wedge Q(t)]]$	H
64	$P(t')$	H
65	$\exists t\,[t' - 1 < t \leq t' \wedge P(t)] \rightarrow \exists t\,[t' - 1 < t \leq t' \wedge Q(t)]$	63
66	$t' - 1 < t' \leq t' \wedge P(t')$	$F, 64$
67	$\exists t\,[t' - 1 < t \leq t' \wedge P(t)]$	66
68	$\exists t\,[t' - 1 < t \leq t' \wedge Q(t)]$	65, 67
69	$t' - 1 < t \leq t' \wedge Q(t)$	H
70	$t = t'$	$69, F$
71	$Q(t')$	69, 70
72	$Q(t')$	$68, 69 - 71$
73	$\forall t\,[P(t) \rightarrow Q(t)]$	$64 - 72$
74	$\forall t\,[P(t) \rightarrow Q(t)] \leftrightarrow$ $\forall t_1, t_2\,[\exists t\,[t_1 < t \leq t_2 \wedge P(t)] \rightarrow \exists t\,[t_1 < t \leq t_2 \wedge Q(t)]]$	$54 - 62, 63 - 73$

B.4 Shared Timepoint Overlap Equivalence

This proof relates a formula asserting the existence of a timepoint that is shared between two intervals, and a simpler timepoint relation that indicates interval overlap.

Theorem

$$\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4] \leftrightarrow t_2 > t_3 \wedge t_1 < t_4$$

Proof
Negating both hand sides provides an alternative formulation of the theorem in terms of *non-overlap*:

$$\neg\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4] \leftrightarrow t_2 < t_3 \vee t_2 = t_3 \vee t_1 = t_4 \vee t_1 > t_4$$

Of Allen's 13 primitive interval relations [1], only the four cases displayed in Figure B.1 below share no point in common between the two intervals $(t_1, t_2]$ and $(t_3, t_4]$, as required by the left hand side of the theorem. As can be seen by inspection, these cases correspond exactly to the disjuncts on the right hand side of the theorem.

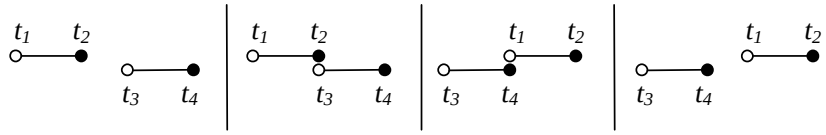

Figure B.1: The four primitive interval relations where the two intervals share no common timepoint.

An alternative, formal deductive, proof is provided below.

75	$\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4]$	H
76	$t_1 < t \leq t_2 \wedge t_3 < t \leq t_4$	H
77	$t_3 < t \wedge t \leq t_2$	76
78	$t_2 > t_3$	$77, F$
79	$t_1 < t \wedge t \leq t_4$	76
80	$t_1 < t_4$	$79, F$
81	$t_2 > t_3 \wedge t_1 < t_4$	$78, 80$
82	$t_2 > t_3 \wedge t_1 < t_4$	$75, 76 - 81$

83	$t_2 > t_3 \wedge t_1 < t_4$		H
84	$t_4 \leq t_2 \vee t_4 > t_2$		F
85	$t_4 \leq t_2$		H
86	$t_1 < t_4 \leq t_2$		$83, 85$
87	$t_3 < t_4 \leq t_4$	t_3 and t_4 are endpoints of an interval	
88	$\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4]$		$86, 87$
89	$t_4 > t_2$		H
90	$t_1 < t_2 \leq t_2$	t_1 and t_2 are endpoints of an interval	
91	$t_3 < t_2 < t_4$		$83, 89$
92	$t_3 < t_2 \leq t_4$		$91, F$
93	$\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4]$		$90, 92$
94	$\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4]$		$84, 85 - 88, 89 - 93$
95	$\exists t\,[t_1 < t \leq t_2 \wedge t_3 < t \leq t_4] \leftrightarrow t_2 > t_3 \wedge t_1 < t_4$		$75 - 82, 83 - 94$

Bibliography

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] James F. Allen. Planning as temporal reasoning. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *Principles of Knowledge Representation and Reasoning (KR'91)*, pages 3–14. Morgan Kaufmann, San Mateo, California, 1991.

[3] Stephen Cresswell, Alan Smaill, and Julian Richardson. Deductive synthesis of recursive plans in linear logic. In *Proceedings of the 5th European Conference on Planning (ECP'99)*, pages 252–264, London, UK, 2000. Springer-Verlag.

[4] Lucas Dixon, Alan Smaill, and Alan Bundy. Planning as deductive synthesis in intuitionistic linear logic. Technical report, The University of Edinburgh Shool of informatics, 2006. `http://homepages.inf.ed.ac.uk/ldixon/papers/infrep-06-planill.pdf`.

[5] Patrick Doherty. PMON$^+$: a fluent logic for action and change, formal specification, version 1.0. Technical Report LiTH-IDA-R-96-33, Department of Computer and Information Science, Linköping University, 1996. `http://www.ida.liu.se/publications/techrep/96/tr96.html`.

[6] Patrick Doherty, Patrik Haslum, Fredrik Heintz, Torsten Merz, Per Nyblom, Tommy Persson, and Björn Wingman. A distributed architecture for autonomous unmanned aerial vehicle experimentation. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, 2004.

[7] Patrick Doherty and Jonas Kvarnström. TALplanner: A temporal logic based planner. *AI Magazine*, 22(3):95–102, 2001.

[8] Patrick Doherty and Jonas Kvarnström. *Handbook of Knowledge Representation*, chapter 18. Elsevier, 2007. To appear.

[9] Alberto Finzi, Fiora Pirri, and Ray Reiter. Open world planning in the situation calculus. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI'00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI'00)*, pages 754–760, Menlo Park, CA, July 30– 3 2000. AAAI Press.

[10] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, October 1998.

[11] Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[12] Joakim Gustafsson. *Extending Temporal Action Logic*. PhD thesis, Linköping University, 2001. Dissertation No. 689.

[13] Koen Hindriks, Yves Lespérance, and Hector Levesque. An embedding of ConGolog in 3APL. Technical Report UU-CS-2000-13, Department of Computer Science, University Utrecht, 2000.

[14] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[15] Franz Inc. Graphical user interface tools. `http://www.franz.com/products/gui_tools/`. Visited November 2006.

[16] Lars Karlsson. Anything can happen: On narratives and hypothetical reasoning. In *Proceedings of the Sixth International Conference on the Principles of Knowledge Representation and Reasoning (KR'98)*, pages 36–47, 1998.

[17] Lars Karlsson. *Actions, Interactions and Narratives*. PhD thesis, Linköping University, 1999. Dissertation No. 593.

[18] Jonas Kvarnström. VITAL: Visualization and implementation of temporal action logics. `http://www.ida.liu.se/~jonkv/vital/`. Visited January 2007.

[19] Jonas Kvarnström. *TALplanner and Other Extensions to Temporal Action Logic*. PhD thesis, Linköping University, 2005. Dissertation No. 937.

[20] Hector J. Levesque. Planning with loops. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 509–515, 2005.

[21] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[22] Vladimir Lifschitz. Circumscription. In *Handbook of Artificial Intelligence and Logic Programming*, volume 3, pages 297 – 352. Oxford University Press, 1991.

[23] Vladimir Lifschitz. Answer set planning. In *Proceedings of the 1999 International Conference on Logic Programming*, pages 23–37, 1999.

[24] Martin Magnusson and Patrick Doherty. Deductive planning with temporal constraints using TAL. In *Proceedings of the International Symposium on Practical Cognitive Agents and Robots (PCAR'06)*, pages 141–152, 2006.

[25] Martin Magnusson and Patrick Doherty. Deductive planning with temporal constraints. In Eyal Amir, Vladimir Lifschitz, and Rob Miller, editors, *Logical Formalizations of Commonsense Reasoning: Papers from 2007 AAAI Spring Symposium*. AAAI Press, 2007. Technical Report SS-07-05 `http://www.ucl.ac.uk/commonsense07/papers/magnusson-and-doherty.pdf`.

[26] Zohar Manna and Richard J. Waldinger. How to clear a block: A theory of plans. *Journal of Automated Reasoning*, 3(4):343–377, 1987.

[27] Sheila McIlraith and Ronald Fadel. Planning with complex actions. In *Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning (NMR'02)*, pages 356–364, Toulouse, France, April 19-21 2002.

[28] Itay Meiri. Combining qualitative and quantitative constraints in temporal reasoning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 260–267, 1991.

[29] Nils J. Nilsson. Eye on the prize. *AI Magazine*, 16(2):9–17, 1995.

[30] Swedish Institute of Computer Science SICS AB. SICStus prolog homepage. `http://www.sics.se/sicstus/`. Visited November 2006.

[31] Francis Jeffry Pelletier. A brief history of natural deduction. *History and Philosophy of Logic*, 20:1–31, 1999.

[32] Erik Sandewall. *Features and Fluents: The Representation of Knowledge about Dynamical Systems*, volume 1. Oxford University Press, 1994.

[33] Murray Shanahan. An abductive event calculus planner. *Journal of Logic Programming*, 44(1-3):207–240, 2000.

[34] Murray Shanahan and Mark Witkowski. High-level robot control through logic. *Lecture Notes in Computer Science*, 1986, 2001.

[35] Tran Cao Son, Chitta Baral, and Sheila McIlraith. Extending answer set planning with sequence, conditional, loop, non-deterministic choice, and procedure constructs. In *Proceedings of the AAAI Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 202–209, Stanford, ca, USA, March 26-28 2001.

[36] Michael Thielscher. FLUX webpage. `http://www.fluxagent.org/`. Visited December 2006.

[37] Michael Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4–5):533–565, 2005.

[38] John Thornton, Matthew Beaumont, Abdul Sattar, and Michael J. Maher. A local search approach to modelling and solving interval algebra problems. *Journal of Logic and Computation*, 14(1):93–112, 2004.

[39] Wikipedia. Presburger arithmetic — wikipedia, the free encyclopedia, 2007. Visited January 2007.

Avdelning, Institution
Division, Department

AIICS,
Dept. of Computer and Information Science
581 83 Linköping

Datum
Date

2007-09-28

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _____

Rapporttyp
Report category

☒ Licentiatavhandling
☐ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
978–91–85895–93–9

ISRN
LiU-Tek-Lic–2007:38

Serietitel och serienummer **ISSN**
Title of series, numbering 0280–7971

Linköping Studies in Science and Technology

Thesis No. 1329

URL för elektronisk version
http://www.martinmagnusson.com/publications/magnusson-2007-lic.pdf

Titel
Title

Deductive Planning and Composite Actions in Temporal Action Logic

Författare
Author

Martin Magnusson

Sammanfattning
Abstract

Temporal Action Logic is a well established logical formalism for reasoning about action and change that has long been used as a formal specification language. Its first-order characterization and explicit time representation makes it a suitable target for automated theorem proving and the application of temporal constraint solvers. We introduce a translation from a subset of Temporal Action Logic to constraint logic programs that takes advantage of these characteristics to make the logic applicable, not just as a formal specification language, but in solving practical reasoning problems. Extensions are introduced that enable the generation of action sequences, thus paving the road for interesting applications in deductive planning. The use of qualitative temporal constraints makes it possible to follow a least commitment strategy and construct partially ordered plans. Furthermore, the logical language and logic program translation is extended with the notion of composite actions that can be used to formulate and execute scripted plans with conditional actions, non-deterministic choices, and loops. The resulting planner and reasoner is integrated with a graphical user interface in our autonomous helicopter research system and applied to logistics problems. Solution plans are synthesized together with monitoring constraints that trigger the generation of recovery actions in cases of execution failures.

Nyckelord
Keywords

Temporal Action Logic, deductive planning, composite actions, interval algebra, constraint logic programming, execution monitoring